Technische Universität Darmstadt

Fachbereich Theoretische Informatik

Prof. Dr. J. Buchmann

Diplomarbeit

Design und Entwicklung eines Testframeworks für JLiPSD

Autor:
Jochen Hähnle

Betreuer:
Dr. T. Setz

Darmstadt, März 2003

Inhaltsverzeichnis

1 Einleitung und Überblick **8**

2 Softwareentwicklung **10**

2.1 Motivation 10

2.2 Definition von Software 10

2.3 Qualitätsmerkmale 11

2.3.1 Qualitätsmerkmale der Dokumentation 11

2.3.2 Qualitätsmerkmale der Software 12

2.4 Software Qualitätssicherung 12

2.4.1 Nutzen 13

2.4.2 Notwendigkeit 13

2.4.3 Ziele 13

2.4.4 Software Lebenszyklus 14

2.5 Maßnahmen der Qualitätssicherung 14

2.5.1 Konstruktive Maßnahmen 15

2.5.2 Analytische Maßnahmen 15

2.6 Softwaremetriken 15

2.6.1 Der kosten- nutzenbezogene Ansatz 16

2.6.2 Der prozeßbezogene Ansatz 16

2.6.3 Der produktbezogene Ansatz 16

3 Softwaretesten **17**

3.1 Stellung in der Softwareentwicklung 17

3.2 Zielsetzung des Softwaretestens 18

3.3 Erfolgreiche und Erfolglose Tests 18

3.4 Operationales und Systematisches Testen 19

3.5 Einschränkungen 20

3.5.1 Vollständiges Testen 20

3.5.2 Partielles Testen 21

3.5.3 Top Down Testen 21

3.5.4 Bottom Up Testen 22

3.6 Stellung im Software Lebenszyklus 22

3.7 Teststrategien 24

3.7.1 Ideale Fehlerbedingungen 24

3.7.2 Systemskalierung 25

3.8 Testverfahren 27

3.8.1 Funktionale Testverfahren 29

3.8.2 Strukturelle Testverfahren 30

3.9 Verläßlichkeitstesten 32

3.9.1 Leistungstests 32

3.9.2 Lasttests 32

3.9.3 Streßtests 32

3.9.4 Reinraumtests 33

3.9.5 Regressionstests 33

3.10 Review Meetings 34

4 Testen objektorientierter Software 35

4.1 Besonderheiten der Objektorientierung 35

4.1.1 Kapselung 35

4.1.2 Vererbung 36

4.1.3 Polymorphie 37

4.2 Komplexitätsvergleich gegenüber prozeduralem Testen 37

4.3 Reflection Testen versus manuelles Testen 38

4.4 Extreme Programming 39

4.5 Anwendbarkeit traditioneller Testmethodik 41

4.5.1 Reviews Meetings 42

4.5.2 Black- Box Testen 42

4.5.3 White- Box Testen 42

4.6 Teststufen 43

5 Anwendung auf JLiPSD 45

5.1 Definition des Testframeworks 45

5.2 Einführung in JLiPSD 46

5.3 Entwicklungsumgebung von JLiPSD 46

5.3.1 JEdit 46

5.3.2 Ant 47

5.3.3 XML 47

5.4 Orientierung 48

5.4.1 Projekteinschätzung 48

5.4.2 Abgrenzung 48

5.5 Untersuchung zur Verfügung stehender Werkzeuge 49

5.5.1 JUnit 49
5.5.2 NoUnit 49
5.5.3 Quilt 50
5.5.4 Clover 50
5.5.5 Gretel 51
5.5.6 Jester 51

6 Design des Testframeworks 53

6.1 Entwurf eines Testprozesses 53
6.1.1 Vorgehensweise 53
6.1.2 Erstellen der Testdokumente 54
6.1.3 Ablauf des Testprozesses 54
6.2 Metriken 55
6.3 Elemente der Testbeschreibung 55
6.3.1 Inhalte eines Testfalls 56
6.3.2 Vertreterfunktionalität 56
6.3.3 Plazierung einer Testfallbeschreibung 56
6.3.4 Absehbare Einschränkungen 57
6.4 Testmuster für Klassentests nach Binder 57
6.4.1 Nonmodale Klassen 57
6.4.2 Unimodale Klassen 58
6.4.3 Quasimodale Klassen 58
6.4.4 Modale Klassen 58
6.4.5 Unterklassen 59
6.5 Testmuster für Integrationstests nach Binder 60

7 Entwicklung **61**

7.1 Werkzeuge des Testframeworks 61

7.1.1 Auswahl eines Klassentestwerkzeugs 61

7.1.2 Auswahl eines Überdeckungswerkzeugs 64

7.2 Das Testframework für JLiPSD 65

7.2.1 Erstellen der Testsuites 66

7.2.2 Konzept des Bedienungsablaufs 68

7.3 Techniken zur Entwicklung guter Testfälle 69

7.3.1 Verwendung innerer Testklassen 69

7.3.2 Behandlung von Ausnahmesituationen 69

7.3.3 Testfälle klein halten 70

7.3.4 Vermeidung von Seiteneffekten 70

7.3.5 Zeitunabhängigkeit 71

7.3.6 Architekturunabhängigkeit 71

7.3.7 Systemtests 71

7.3.8 Selbstbeschreibende Namensgebung 71

8 Das Testframework in der Anwendung **74**

8.1 Bedienung des Testframeworks 74

8.1.1 Einfügen eines Testfalls 75

8.1.2 Einfügen einer Testbeschreibung 75

8.1.3 Erstellen eines Klassentests 76

8.1.4 Aktualisieren der Testsuites 76

8.1.5 Testausführung 77

8.1.6 Konsultieren der Ergebnisdaten 78

8.1.7 Wiederholung von Testläufen 79

8.2 Testergebnisse von JLiPSD 80

8.2.1 Überdeckungsergebnisse 80

8.2.2 Analyse entdeckter Fehler 81

9 Zusammenfassung und Ausblick **90**

Abbildungsverzeichnis

3.1 Das V-Modell des Software Lebenszyklus 23

3.2 Das V-Modell zur Planung und Durchführung der Testaktivitäten . . 28

4.1 Ein Beispiel zum dynamischen Binden 37

6.1 Zustandsübergangsgraph einer Warteschlangen Klasse 59

7.1 Klassendiagramm der wichtigsten JUnit Klassen 62

7.2 JUnit Swing-Ergebnisreport 63

7.3 Clover Swing Ergebnisreport 65

7.4 Die Test Hauptsuite des JLiPSD 67

7.5 Die Untersuite der FileTransfer Package 68

7.6 Unittest Beispiel der Klasse ByteConverter 72

7.7 Exceptionbehandlung, wenn diese abgefangen werden soll 73

7.8 Exceptionbehandlung, wenn diese nicht auftritt 73

7.9 Beispiel zur Testnamensgebung 73

8.1 Platzierung der Testbeschreibung 75

8.2 Beispiel Testsuite für die tools Package 76

8.3 JEdit mit integriertem Ant-Werkzeug 84

8.4 JUnit : Swing Ergebnisreport 85

8.5 Clover Swing Report 86

8.6 Clover HTML/XML Report - Klassenansicht 87

8.7 Clover HTML/XML Report - Testsuiteansicht 88

8.8 Clover Überdeckungsergebnis des JLiPSD Projektes 89

Kapitel 1

Einleitung und Überblick

LiPS, Library for Parallel Systems, ist ein System zur verteilten Berechnung auf UNIX Arbeitsplatzrechnern. Das Projekt wird seit 1992 am Lehrstuhl von Prof. Buchmann von Herrn Dr.-Ing. Thomas Setz geleitet. Das System soll in den kommenden Jahren weiter ausgebaut werden, um Anwendungen auf mehr als 1000 weltweit verteilten Rechnern zu ermöglichen.

Hierfür wurde der LiPSD (Library for Parallel Systems Daemon), welcher in der Programmiersprache C implementiert ist, nach Java portiert (JLiPSD), damit LiPS plattformunabhängig betrieben werden kann. Bei der Portierung des LiPSD in die objektorientierte Programmiersprache Java wurden bestimmte Anforderungen sowohl an das Systemdesign, als auch an die Softwareentwicklung gestellt.

Ein zentrales Gebiet in der Softwareentwicklung ist die Softwarequalität. Diese ist erreicht, wenn ein Programm die Anforderungen an die Software unter realistischen Anwendungssituationen (einschließlich Ausnahmesituationen) erfüllt. Dies zu erreichen ist keineswegs trivial, da für Software allein schon die Fehlerfreiheit nicht nachgewiesen werden kann. Um jedoch ein hohes Maß an Fehlerfreiheit zu erreichen, stellt die Softwarequalitätssicherung konstruktive sowie analytische Maßnahmen bereit. Ein Teil dieser Maßnahmen ist das Softwaretesten, welches zu den kostenintensivsten Teilbereichen in der Qualitätssicherung zählt. Konsequent durchgeführtes Testen resultiert jedoch in Fehlererkennung, Fehlerbehebung und Fehlervorhersage, wodurch Softwarequalität in wesentlichem Maße gesteigert wird.

Ziel dieser Diplomarbeit ist es mittels eines durchdachten Designs ein Java Testframework für JLiPSD zu entwickeln, welches ein hohes Maß an Softwarequalität erreicht.

In dem folgenden Kapitel wird der Leser in das Gebiet der Softwareentwicklung eingeführt und mit den wesentlichen Elementen vertraut gemacht, soweit sie für diese Arbeit notwendig sind. Im darauf folgenden Kapitel werden Methoden und Verfahren des Softwaretestens vorgestellt. Hierbei wird auch auf die Terminologie und auf verschiedene Vorgehensweisen des Softwaretestens eingegangen, welche für die Entwicklung und das Design eines Testframeworks benötigt werden. Daran anschließend wird das Testen von objektorientierter Software beleuchtet. Hierbei wird das Extreme Programming (XP) Paradigma erklärt sowie aufgezeigt, welche traditionellen Testmethoden weiter verwendet werden können und welche Besonderheiten bei der Objektorientierung berücksichtigt werden müssen. Dieses Wissen wird im nächsten Kapitel dazu benutzt, um ein Testframework für JLiPSD zu definieren und geeignete Werkzeuge hierfür zu untersuchen. Im sechsten Kapitel wird das Design des Testframeworks erstellt, wofür die Metriken, die Elemente der Testbeschreibung sowie einige Testpatterns von Binder vorgestellt werden. Im siebten Kapitel wird das Testframework entwickelt, wozu Werkzeuge ausgewählt werden. Desweiteren wird die Testsuite erstellt sowie auf Techniken zur Entwicklung guter Unittests eingegangen. In Kapitel acht wird die Anwendung des Testframeworks erläutert. Dabei wird sowohl auf die Bedienung des Testframeworks eingegangen. Zusätzlich werden die Gesamttestergebnisse des JLiPSD bewertet. Die Arbeit wird durch Zusammenfassung und Ausblick abgeschlossen.

Viele Begriffe aus der Terminologie des Softewaretestens sind nur in englischer Sprache vorhanden. Wo immer es möglich ist, wird eine deutsche Übersetung verwendet. Um die Zuordnung von deutscher und englischer Terminologie zu erleichtern, wird jeder Begriff zweisprachig eingeführt, falls eine deutsche Übersetzung hierzu existiert.

Kapitel 2

Softwareentwicklung

Die Softwareentwicklung unterteilt sich grob in zwei Teilgebiete. Das eine Teilgebiet ist die Softwaretechnik (Software engineering), das andere ist die Softwarequalitätssicherung (Software Quality assurance). In diesem Kapitel wird ein Überblick über die verschiedenen und für diese Arbeit wichtigen Teilaspekte des Gebiets der Softwareentwicklung gegeben. Hierbei werden die für kommende Kapitel wichtigen Begriffe eingeführt. Die nachfolgende Motivation soll dem Leser einen Einblick über Notwendigkeit und Zielsetzung der Softwareentwicklung verschaffen.

2.1 Motivation

Pannen in der Software kosten Zeit, Stress und Geld. In so manchem Softwareprojekt stecken zündende Ideen: Ein falscher Befehl und das ehrgeizige Projekt endet in einem Feuerwerk. Häufiger als man vermutet stehen hierbei Millionen auf dem Spiel. Einige der bekanntesten Beispiele sind: der Absturz der Ariane-5-Rakete beim Jungfernflug, die Airbus A320 Bruchlandung in Moskau (wegen nicht überbrückbarer Computer), ca. 6 Millionen Fehler in Windows NT 5.0 unmittelbar nach erscheinen der Software ([1]).

2.2 Definition von Software

Unter Software werden nach IEEE (Institute of Electrical and Electronics Engineers) Standard nicht nur Daten und Programme (=Daten und Algorithmen) sondern auch die dazugehörige Dokumentation verstanden. Es gibt zwei Arten der Dokumentation. Benutzungsdokumentation und Entwicklungsdokumentation. Diese werden nachfolgend vorgestellt ([2]).

Benutzungsdokumentation

Die Benutzungsdokumentation als Teil der Software setzt sich aus Entscheidungsinformation, Benutzungsanleitung sowie Maschinenbedarfs- und Maschinenbedienungsanweisungen zusammen ([2] S. 23-24).

Entwicklungsdokumentation

Die Entwicklungsdokumentation enthält alle Anweisungen für die Entwickler, Kontrolleure und das Wartungspersonal eines Softwaresystems. Es setzt sich zusammen aus :

- Systemspezifikation bzw. Pflichtenheft (Anwendungsentwurf)
- Entwurf (Technischer Entwurf und Lösungskonzept)
- Programmdokumentation
- Testdokumentation

Weitere Informationen hierzu finden sich unter ([2] S. 24-25).

2.3 Qualitätsmerkmale

Softwarequalität bezieht sich immer auf die oben genannten Komponenten von Software, nämlich die Benutzungs- und Entwicklungsdokumentation sowie die Programme und Kommandoprozeduren ([2] S. 27).

2.3.1 Qualitätsmerkmale der Dokumentation

Die Qualitätsmerkmale der Dokumentation bestehen aus:

- Verständlichkeit - Die Dokumentation muß für die Zielgruppe verständlich sein.
- Reproduzierbarkeit - Die Reproduzierbarkeit erfordert, daß das gewählte Verfahren zur Ermittlung und Bewertung eines Qualitätsmerkmals wiederholbar ist und dabei die gleichen Ergebnisse wie vorher erbringt.

- Veränderbarkeit - Da sich die Grundlage der Dokumentation ändern kann, muß es möglich sein die Dokumentation anpassen zu können.
- Überschaubarkeit -Eine Dokumentation sollte eine übersichtliche Gliederung und Struktur aufweisen.
- Genauigkeit - Es ist auf Genauigkeit und Konsistenz zu achten, um die Dokumentation fehlerfrei zu halten.
- Widerspruchsfreiheit - Eine Dokumentation muß eindeutig und somit widerspruchsfrei gehalten sein.
- Vollständigkeit - Um Verständnisschwierigkeiten zu vermeiden, müssen die Dokumente vollständig sein.

Weitere Informationen bezüglich der Qualitätsmerkmale finden sich unter ([2] S. 27-28).

2.3.2 Qualitätsmerkmale der Software

Als Hauptziele von Software können die Benutzerakzeptanz und die Ausbaufähigkeit gesehen werden. Wenn die Erwartungen des Benutzers nicht erfüllt werden, ist die Software zum scheitern verurteilt, egal wie gut sie konstruiert ist. Ebenso muss sie weiterentwicklungsfähig sein, um das System über längere Zeit hinweg verwenden zu können ([2] S. 28-31).

2.4 Software Qualitätssicherung

Unter der Qualitätssicherung versteht man die Gesamtheit der Tätigkeiten der Qualitätsplanung, -lenkung und -prüfung.

- Qualitätsplanung: Auswahl, Klassifikation und Gewichtung der Qualitätsmerkmale sowie Festlegung der Qualitätsanforderungen unter Berücksichtigung der Realisierungsmöglichkeiten.
- Qualitätslenkung: Überwachen und Steuern der Realisierung einer Einheit mit dem Ziel, die Qualitätsanforderung zu erfüllen.
- Qualitätsprüfung: Überprüfen, inwieweit eine Einheit die Qualitätsanforderungen erfüllt.

Diese allgemeine Definition von Qualitätssicherung ist auch für die Softwarequalitätssicherung gültig ([3] S. 281-282).

2.4.1 Nutzen

Die durch Softwarefehler verursachten tödlichen Unfälle sind zwar eher selten, aber dennoch zu beachten. Weil ein entdeckter Fehler in den Flugkursdaten dem Piloten nicht übermittelt wurde, stürzte 1979 ein Passagierflugzeug mit 250 Personen an Bord ab. Zu enormen finanziellen Verlusten und chaotischen Zuständen führten Fehler in Bankensystemen und Telefonvermittlungssystemen in Nordamerika.
In den USA rechnete man im Jahr 1996 mit 8-10 Fehlern pro 1000 Zeilen Programmcode. Diese Fehlerrate hätte sich durch Anwendung relativ einfacher Softwarequalitätssicherungsmethoden um einen Faktor von nahezu 100 reduzieren lassen
([3] S. 15).

2.4.2 Notwendigkeit

Wenn besonderer Wert auf Qualität gelegt wird sind unabhängige Überprüfungen notwendig um sicherzustellen, daß den Menschen bei Ihrer Arbeit kein Fehler unterlaufen ist. Die Fragen, die sich im Zusammenhang mit Software stellen sind nicht, ob Überprüfungen durchgeführt werden sollen oder nicht, sondern wer diese durchführt und vor allem wie. In kleineren Unternehmen ist es meist möglich, daß die Softwaremanager die Arbeit selbst überwachen und daher keine Notwendigkeit für die Softwarequalitätssicherung besteht. Sobald ein Unternehmen aber größer ist, verändert sich der Tätigkeitsbereich der Manager. Es bleiben ihnen dann folgende Möglichkeiten:

- Es wird jemand eingestellt, der die Kontrolle übernimmt.
- Motivation der Mitarbeiter, sich selbst zu kontrollieren.

([4] S. 139)

2.4.3 Ziele

Die Ziele der Qualitätssicherung sind:

- Verbesserung der Softwarequalität durch geeignete Überwachung der Software und des Softwareentwicklungsprozesses.
- Gewährleistung der Einhaltung von bestehenden Standards und Methoden für die Software und den Softwareentwicklungsprozess.

- Sicherstellung, daß jegliche Inadäquatheit im Produkt, im Prozess oder den Standards dem Management mitgeteilt wird und dadurch beseitigt werden kann.
- Softwarequalitätssicherung bzw. die Personen die diese durchführen, sind nicht für die Herstellung von Qualitätsprodukten oder die Entwicklung von Qualitätsplänen verantwortlich. Die Softwarequalitätssicherung ist ausschließlich für die Prüfung der Qualitätsmaßnahmen und die Information des Managements bei etwaigen Abweichungen zuständig.
- Um wirklich effektiv zu sein, muß die Softwarequalitätssicherung eng mit der Softwareentwicklung zusammenarbeiten. Sie muß mit den Plänen vertraut gemacht sein, ihre Ausführung überprüfen und die Durchführung der einzelnen Aufgaben überwachen. Dabei dürfen die Entwickler die Softwarequalitätssicherung aber nie als ihren Feind betrachten.

Eine detaillierte Beschreibung findet sich unter ([4] S. 140).

2.4.4 Software Lebenszyklus

Die Sicherung der Softwarequalität hängt eng mit dem Softwareentwicklungsprozeß zusammen, der zyklisch verläuft. Man spricht deshalb auch vom *Software Lebenszyklus* (Software Life Cycle), einem zentralen Begriff in der Softwareentwicklung. "Voraussetzung für eine systematische Softwarequalitätssicherung ist die Lebenszyklus - Entwicklungsstrategie, wonach die Software ständig weiterentwickelt wird"([2] S. 65).
Der Software Lebenszyklus ist ein phasenorientiertes Vorgehensmodell. Jede Phase führt zu einem wohledefinierten Zwischenergebnis bzw. Zwischenprodukt, welches als Eingabe für die nächste Phase verwendet wird. Es wird nicht nur das Zwischenprodukt selbst, sondern auch seine Qualitätsmerkmale in Form von Richtlinien und Namen definiert, wodurch das Softwareprodukt gemessen werden kann. Durch diesen dauernden Soll/Ist Vergleich erhält das Management die Möglichkeit immer wieder steuernd in den Softwareentwicklungsprozeß einzugreifen. Zur Sicherung der Softwarequalität müssen die Zwischenprodukte ständig geprüft werden ([2] S. 65-66).

2.5 Maßnahmen der Qualitätssicherung

Grundsätzlich unterscheidet man in der Qualitätssicherung zwischen konstruktiven und analytischen Maßnahmen. Diese werden nachfolgend vorgestellt.

2.5.1 Konstruktive Maßnahmen

Die konstruktiven Maßnahmen sind Verfahren in der Anforderungsphase, im Entwurf und in der Implementierung zur Fehlervermeidung. Erstes Ziel einer konstruktiven Maßnahme ist die Fehlervermeidung bzw. Fehlerverminderung. Das zweite Ziel ist, nicht verhütete Fehler schnell zu entdecken, ihren Schaden zu begrenzen und sie schnell und effektiv zu beheben. Die Verfahren zur Fehlervermeidung ziehen sich durch alle Entwicklungsphasen und sind entsprechend den Phasen unterschiedlich geartet ([3] S. 103).

2.5.2 Analytische Maßnahmen

Die analytischen Maßnahmen können in Inspektion, Analyse, Testen und funktionale Verifikation unterteilt werden. Das Ziel einer analytischen Maßnahme ist es, folgende Punkte sicherzustellen:

- Erreichen der geforderten Qualitätsmerkmale
- Durchführen der dazu notwendigen konstruktiven Maßnahmen
- Fehler im Softwarepaket finden und beheben

Es ist dabei zu beachten, daß qualitativ angehbare Qualitätsmerkmale (z.B. Änderbarkeit) sich auch nur qualitativ bewerten lassen. Quantitative Qualitätsmerkmale dagegen lassen sich messen bzw. abschätzen ([3] S. 153).

2.6 Softwaremetriken

Insbesondere für sicherheitsrelevante Systeme ist es von enormer Bedeutung, daß die entwickelte Software den Qualitätsanforderungen entspricht. Dies wird durch intensives Testen geprüft. Doch wie wird entschieden, ob die Software ausreichend zuverlässig, effizient oder wartbar ist? Hier helfen dem Tester Metriken, die ihm Maße zur Planung, Steuerung und Kontrolle der jeweiligen Testphase zur Hand gehen. Aber auch andere Aspekte von Qualität wie beispielsweise Faktoren, die zur Sicherstellung eines akzeptablen Preis-/Leistungsverhältnisses beitragen, können mit Hilfe von Metriken erfaßt werden.
Ein weiterer Vorteil ist, daß Metriken die Chance bieten, die gewünschte Qualität des Softwaretests und damit auch die Produktqualität nachweisbar sicherzustellen. Zusätzlich werden erreichte Verbesserungen leichter nachvollziehbar. Hierbei gibt es mehrere Ansätze die nachfolgend aufgeführt werden.

2.6.1 Der kosten- nutzenbezogene Ansatz

"Qualität ist eine Funktion von Kosten und Nutzen. Ein Qualitätsprodukt ist ein Erzeugnis, das einen bestimmten Nutzen zu einem akzeptablen Preis erbringt".
([5] S. 256)

Diese Begriffsbildung basiert auf den Worten Kosten und Nutzen, d.h. daß in diesem Unterpunkt speziell auf Metriken eingegangen wird, welche sich auf die Kostenentwicklung der Software beziehen oder dem Auftraggeber sicherstellen, daß die gewünschte Funktionalität vorhanden ist.

2.6.2 Der prozeßbezogene Ansatz

"Qualität entsteht durch die richtige Erstellung des Produkts. Der Erstellungsprozeß wird spezifiziert und kontrolliert, um Ausschuß- und Nachbearbeitungskosten zu reduzieren (right the first time) und um ihn permanent an sich wandelnde Kundenbedürfnisse zu adaptieren". ([5] S. 256).

Im Falle der Softwareentwicklung gehört zur richtigen Erstellung das Testen. Zuerst werden Metriken eingeführt welche helfen zu bestimmen, wie komplex die Software ist, welche Testverfahren passend bzw. am besten geeignet sind und anschließend wie sie helfen Tests zu steuern und zu kontrollieren. Das heißt beispielsweise bestimmen sie, wann genügend Tests durchgeführt wurden.

2.6.3 Der produktbezogene Ansatz

"Qualität ist eine meßbare, genau spezifizierte Größe, die das Produkt beschreibt und durch die man Qualitätsunterschiede aufzeigen kann. Subjektive Beobachtungen und Wahrnehmungen werden nicht berücksichtigt. Anhand der gemessenen Qualität kann eine Rangordnung von verschiedenen Produkten der gleichen Kategorie aufgestellt werden. Dieser Ansatz bezieht sich nur auf das Endprodukt, nicht auf den Kunden. Das kann zu einer mangelnden Berücksichtigung der Kundeninteressen führen". ([5] S. 256)

Hierfür gibt es einige Metriken, die nichts mit der Entwicklung der Software zu tun haben, sondern diese nur als Endprodukt und komplette Einheit betrachten, z.B. Lines of Code (LOC) (siehe Abs. 6.2).

Kapitel 3

Softwaretesten

Nach Einführung des Softwaretestens und der Einordnung dessen in der Qualitätssicherung werden im folgenden Kapitel die wesentlichen Voraussetzungen auf denen das Softwaretesten aufbaut vorgestellt. Hierzu werden die Grenzen des Testens aufgezeigt und die Planung des Testprozesses durch Einpassung in den Software Lebenszyklus beschrieben. Abschließend werden übliche Strategien sowie Verfahren des Softwaretestens vorgestellt.

3.1 Stellung in der Softwareentwicklung

Verifikation[1] und Validation[2] haben im Qualitätsmanagement der Softwareentwicklung eine zentrale Bedeutung. Sie sind Bestandteile eines geordneten Softwareentwicklungsprozesses und unterliegen somit auch vergleichbaren Einschränkungen in der Verteilung von Ressourcen wie Zeit, Personal oder Equipment. Um Verifikation der Software im Rahmen dieser Einschränkungen zu erreichen, ist es bei Softwareprojekten angemessener Größe und Komplexität üblich, gar keine oder nur kritische Programmteile formal zu verifizieren. Um Software zuverlässiger zu machen, wird diese getestet.
Die Behebung von Softwarefehlern wird umso teurer, je später der Fehler im Softwarelebenszyklus aufgedeckt wird. Empirische Untersuchungen hierzu haben gezeigt, daß zwei Drittel aller gefundenen Fehler in Analyse und Design gemacht werden und nur der Rest in der Implementierung steckt ([6] S. 21).
Bei Softwarefirmen mit einem wohldefinierten Testprozeß verteilen sich die Entwicklungskosten gleichmäßig auf Softwaretesten und Softwareentwicklung.

[1] Mittels Verifikation wird festgestellt, ob ein Programm seiner Spezifikation entspricht. Sie wird im allgemeinen formal realisiert.

[2] Validation ist die Prüfung, ob die Beschreibung eines Algorithmus mit dem zu lösenden Problem übereinstimmt. Sie ist im allgemeinen nicht formal durchzuführen.

3.2 Zielsetzung des Softwaretestens

Unter Softwaretesten versteht man die Verifikation und Validation einer Softwarekomponente oder eines Softwaresystems. Softwaretesten ist nur möglich, wenn Anforderungsdokumente und Spezifikationsdokumente vorliegen. Das Anforderungsdokument bestimmt, was die Software zu leisten im Stande sein muß, um qualitativ hochwertig zu sein. Dies wird oft vom Benutzer vorgegeben. Die Spezifikation besagt, wie die Anforderungen an die Software erfüllt werden.
Der Prozeß des Softwaretestens besteht in dem Bemühen, Abweichungen oder Unvollständigkeiten in der Spezifikation zu finden. Weitergehend werden die in Implementierung oder Spezifikation gefundenen Abweichungen korrigiert und die geänderte Software unter der gegebenenfalls erweiterten Spezifikation erneut getestet. Hierdurch wird die Implementierung verifiziert.
Andererseits stellt man durch Softwaretesten fest, ob Spezifikation und Implementierung den Anforderungen an die Software genügen. Dabei wird auch unterstellt, daß die Spezifikation nicht der Problemstellung entspricht, sie also Fehler wie z.B. Unvollständigkeiten oder Widersprüche enthalten kann. Somit wird die Software validiert.
Hierbei liegt die Vermutung zugrunde, daß die Softwarekomponente möglicherweise nicht das zu leisten vermag, was von der Problemstellung vorgegeben ist. Entwickler und insbesondere Anwender haben kein großes Vertrauen in die Qualität eines ungetesteten Softwareprodukts. Dies soll durch Softwaretesten hergestellt werden. Dabei ist jedoch nicht die Darstellung eines "fehlerfreien"Softwarepakets das Ziel, sondern das Aufzeigen seiner Schwachpunkte. Dies beruht auf der Erkenntnis, daß kein größeres Softwarepaket fehlerfrei sein kann (siehe Abs. 3.5). Softwaretests können nur die Anwesenheit von Fehlern feststellen, niemals deren Abwesenheit.

3.3 Erfolgreiche und Erfolglose Tests

Häufig werden Aussagen benutzt, wie :"Testen ist ein Prozeß der zeigen soll, daß keine Fehler vorhanden sind". Oder: "Der Zweck des Testens ist es zu zeigen, daß ein Programm die geforderten Funktionen korrekt ausführt"([7] S. 3).
Das eigentliche Ziel des Softwaretestens ist jedoch, daß der Tester den Wert des Programms durch das Testen anhebt, was natürlich für die Software bedeutet, daß die Zuverlässigkeit erhöht werden soll. Zuverlässigkeit wird durch das Auffinden von Fehlern erreicht. Eine angemessene Definition des Testens ist daher: "Testen ist der Prozeß, ein Programm mit der Absicht auszuführen, Fehler zu finden". Ein erfolgreicher Test ist somit ein Test, der nicht das erwartete Ergebnis liefert und somit

einen Fehler aufdeckt. Ein erfolgloser Test ist deshalb ein Test, der den gewünschten Zielvorgaben entspricht.
Die Zielvorgaben eines Tests werden von Menschen bestimmt. Da Menschen aber höchst zielorientiert vorgehen, hat die Vorgabe eines Ziels psychologische Effekte, was in diesem Fall bedeutet, daß jemand der mit der Vorgabe an einen Test herangeht, beweisen zu wollen, daß keine Fehler im Programm zu finden sind, weniger effektiv diesen Test durchführen wird. Ein ähnliches Problem tritt auf, wenn ein Programmierer sein eigenes Programm testen soll. Er findet im Schnitt sehr viel weniger Fehler, als ein unabhängiger Tester. Dies gilt ebenso für Programmierteams, die ihr Programm selbst testen wollen. Denn nach Definition ist ein Test ein destruktiver Prozeß und wer versucht schon gerne, nachdem er sein Programm fertiggestellt hat, es mit allen Mitteln zu zerlegen und Fehler zu entdecken.

3.4 Operationales und Systematisches Testen

Wenn man sich z.B. an den Anforderungen eines Testfalls orientiert und diese systematisch durcharbeitet, so spricht man von systematischem Testen. Auf diese Art und Weise werden alle Komponenten der Software gleichermaßen gründlich getestet. Vorteile des systematischen Testens liegen vor allem darin, daß sehr viele Fehler gefunden werden. Der Hauptnachteil dieser Testart liegt darin, daß keinerlei Rücksicht darauf genommen wird, welche Programmstücke hauptsächlich verwendet werden. Es kann auch sein, daß Fehler entfernt wurden die eigentlich nie aufgetreten wären. Wenn dann beim Entfernen dieser Fehler neue ins Programm eingebaut werden, hat man danach ein weniger zuverlässiges Produkt als vorher.
Um feststellen zu können, welche Programmteile beim operationellen Testen getestet werden müssen, muß man wissen wie ein üblicher Programmablauf aussehen wird. Während dies bei gewissen Anwendungen einfach ist (z.B. Telekommunikationssoftware), kann es bei anderen Anwendungen nahezu unmöglich sein ein ordentliches, nützliches Ausführungsprofil zu erstellen. In der Situation in welcher das Ausführungsprofil schwer erstellt werden kann ist operationelles Testen kaum sinnvoll, in anderen Situationen hingegen bzw. in Situationen die irgendwo dazwischen anzuordnen sind, mag operationelles Testen durchaus mächtiger sein als systematisches.
Das Ausführungsprofil für diese Fälle wird dann dadurch erstellt, daß aufgrund des Benutzerverhaltens festgestellt wird, welche Programmteile besonders häufig benutzt werden. Es wird also eine Gewichtung der Programmteile vorgenommen.
Probleme am Ausführungsprofil entstehen, wenn es eine breite Nutzergruppe mit unterschiedlichen Anforderungen gibt oder aber sich das Nutzerverhalten im Laufe

der Zeit ändern sollte. Nach Fertigstellung des Nutzerprofils kann mit dem operationellem Testen begonnen werden.
Vorteile dieser Testart liegen vor allem darin, daß zu Beginn der Testdurchführung jene Fehler entdeckt werden, die am wahrscheinlichsten aufgetreten wären. Am Anfang der Testdurchführung werden also jene Fehler entfernt, die den Programmablauf empfindlich stören, was zu einer schnellen Zuverlässigkeitssteigerung führt.
Die Hauptnachteile liegen darin, daß eventuell ganze Programmstücke nie getestet werden, wenn sie im Ausführungsprofil nicht aufgeführt werden. Es mag also sein, daß fatale Fehler im Programm verbleiben, die bei anderen Teststrategien schnell gefunden werden.
Insgesamt kann man sagen, daß bei operationellem Testen eine asymptotische Annäherung an einen bestimmten Zuverlässigkeitslevel erreicht wird, da am Anfang der Testdurchführung jene Fehler entdeckt werden, die einen hohen Einfluß auf die Zuverlässigkeit haben, und später dieser Einfluß weiter abnimmt.

3.5 Einschränkungen

Das ideale Softwarepaket sollte fehlerfrei sein und alle Anforderungen erfüllen. Hierbei gibt es jedoch leider einige teils empirisch erfaßte Grenzen die es dem Entwickler nicht ermöglichen, sich dieses Ideals zu vergewissern. Keinem Programm kann eine wirkliche Fehlerfreiheit zugewiesen werden. Nachfolgend werden einige grundlegende Strategien zum Testen von Software vorgestellt.

3.5.1 Vollständiges Testen

Vollständiges Testen bedeutet, daß ein Programm mit allen Eingabemöglichkeiten getestet wird. Hierdurch würde man alle Fehler finden. Leider ist es schon unmöglich auch kleinste Teile von Software vollständig zu testen. Die Gründe hierfür sind:

- Zu viele Eingabemöglichkeiten. Wenn z.B. eine Zeichenkette übergeben wird, kann diese beliebig lang sein, von den Variationsmöglichkeiten ganz abgesehen.
- Indeterministisches Verhalten: Einige unsichtbare Eingaben, wie z.B. Nicht vorhersehbare Echtzeit- Interaktionen mit einem Benutzer.
- Die Beurteilung der Ergebnisse selbst kann zu kostenintensiv werden, wenn die Referenzergebnisse beispielsweise vom Menschen per Hand berechnet werden.

Daher ist vollständiges Testen nicht möglich.

3.5.2 Partielles Testen

Beim partiellen Testen wird das Programm nur mit ausgewählten Eingaben getestet. Die Qualität der ausgewählten Eingaben bestimmt somit auch die Effizienz der Tests. Die Eingaben können durch unterschiedlichste Kriterien bestimmt werden, welche sich entweder in Black-Box oder White-Box Tests kategorisieren lassen.

Black- Box Testen

Der Black-Box Test, auch Funktionstest genannt, prüft ein Programm mit konkreten Werten, ob es bei der Transformation der Eingaben in die Ausgaben auch zu der spezifizierten Transformation kommt. Hierbei wird die Software ohne Kenntnis über seine interne Struktur getestet. Black-Box Testen ist daher auch funktionales Testen.

White- Box Testen

Der White-Box Test, auch Strukturtest genannt, prüft ein Programm mit konkreten Werten. Hierbei handelt es sich um eine Analyse der inneren Struktur eines Systems, oder meist einer Systemkomponente. Testdaten werden vom Tester aus der Programmlogik hergeleitet. Dabei ist man bemüht, alle Pfade innerhalb eines Programms einmal auszuführen. Hierbei wird das Programmverhalten als solches getestet. Im Gegensatz zum funktionalen Testen liegt das Augenmerk allein auf der Implementierung. White-Box Testen wird auch als strukturelles Testen bezeichnet und mit Überdeckung gemessen. Testüberdeckung (auch Testabdeckung, siehe Abs. 3.8.2) sind Metriken zur Beurteilung der Testgüte. Mit Überdeckung wird gezählt, wie oft bestimmte Programmteile durch einen Test ausgeführt wurden. Hierbei erhält der Softwaretester einen Überblick über den bereits getesteten sowie ungetesteten Programmcode und kann gegebenenfalls seine Tests erweitern bzw. neue Testfälle hinzufügen.

3.5.3 Top Down Testen

Beim Top-Down Testen werden die Module der obersten Ebenen zuerst entwickelt und getestet. Hierbei ist es jedoch meist äußerst schwer, Vertreterfunktionalitäten (siehe 6.3.2) für Funktionen zu schreiben. Als Vorteil erweist es sich beim Top-Down Testen, daß Fehler in den oberen Ebenen gefunden werden, bevor die unteren Ebenen überhaupt geschrieben worden sind.

3.5.4 Bottom Up Testen

Beim Bottom-Up Testen werden die Module in der Reihenfolge Ihrer Abhängigkeiten geschrieben und getestet. Diese Strategie vereinfacht das Testen. Hierbei werden die einflußreichsten Fehler aber erst gegen Ende der Softwareentwicklung gefunden! Dies kann im schlimmsten Fall zum Verwerfen eines kompletten, fertiggestellten sowie getesteten Moduls führen.

3.6 Stellung im Software Lebenszyklus

Die konventionelle Vorgehensweise des Softwaretestens ist Testen der Software nach ihrer Implementierung. Der Nachteil hierbei ist jedoch, daß nachträgliches Testen sehr kostenintensiv sein kann. Je eher ein Fehler im Softwarelebenszyklus (Software Life Cycle) entdeckt wird, desto geringer sind die Kosten für seine Korrektur. Beim Lebenszyklustesten wird daher das Testen parallel zur Softwareentwicklung durchgeführt.
Die Einordnung des Softwaretestens im Software Lebenszyklus unterliegt dem V-Modell ,wie es aus der VDI von 1993 ([9]) abgeleitet wird und in Abbildung 3.1 zu sehen ist. Die einzelnen Testphasen stehen den einzelnen Entwurfsphasen gegenüber. Die Phasen von der Problemanalyse bis zur Codierung werden hier nicht weiter erläutert, da sie zum Softwareentwurf gehören. Sie sind aufgeführt, um den Zusammenhang zu den einzelnen Testphasen zu verdeutlichen. Spezifikationen, auf die sich die einzelnen Testphasen beziehen, sind durch die schwarzen Pfeile zu den entsprechenden Phasen des Entwurfs gekennzeichnet. Die unterbrochenen Pfeile geben an, dass sich die Testphasen auch auf Informationen aus darüberliegenden Entwurfsphasen stützen. Ein Softwaretest setzt allerdings nicht erst dabei an, die fertigen Komponenten und Module, bzw. das Gesamtsystem zu testen, sondern es müssen schon in den Entwurfsphasen die Zwischenergebnisse nach möglichen Fehlerquellen durchforstet werden, um mögliche Folgefehler und die damit verbundenen Kosten bei einer späteren Beseitigung zu minimieren.
Wie man aus der Abbildung 3.1 erkennen kann, wird die Hardware von der Software ab der Grobentwurfsphase separat entwickelt, um später vor dem Test des integrierten Systems wieder zusammengeführt zu werden. Dies spielt eine wichtige Rolle für den Test des integrierten Systems. Zum Beispiel wird beim Bau eines neuen Schweißroboters die Steuerungssoftware parallel zur Hardwareentwicklung durchgeführt. Zum Schluß soll jedoch die Software zusammen mit dem Roboter funktionieren, und dieses Zusammenspiel muß vor der Übergabe an den Empfänger getestet werden, damit es später nicht zu unschönen Erlebnissen kommt.

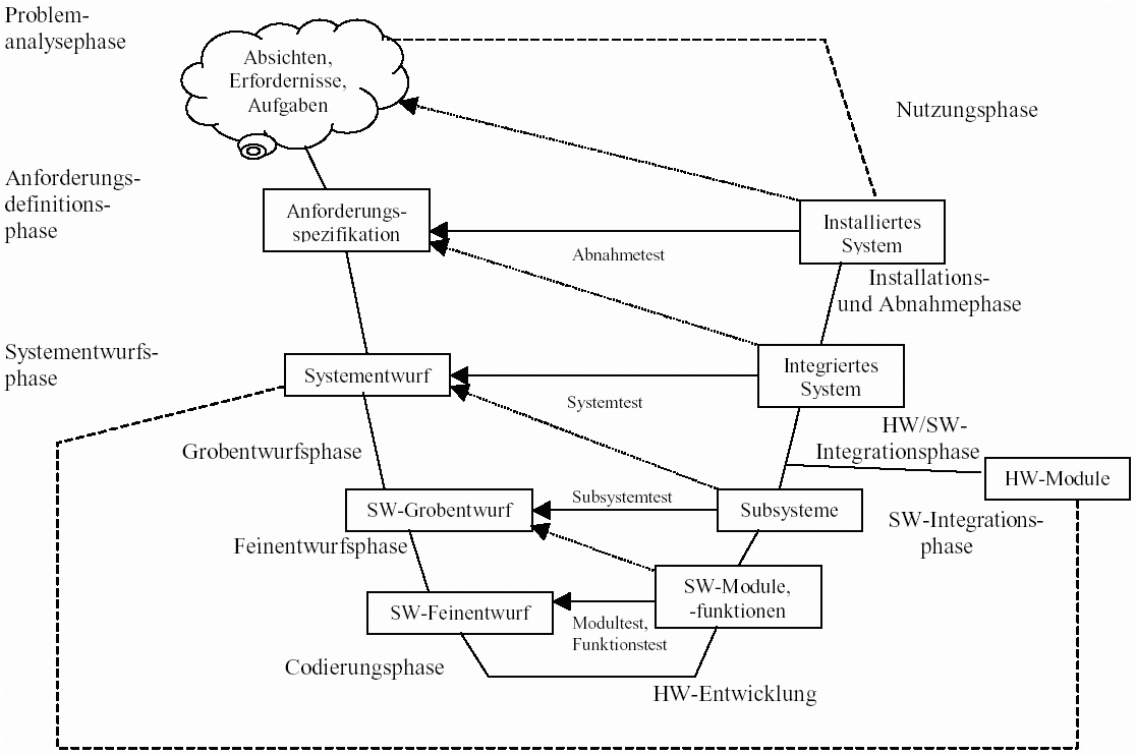

Abbildung 3.1: Das V-Modell des Software Lebenszyklus

Die Testphasen lassen sich in den Funktions- und Modultest, den Test der Subsysteme, den Test des integrierten Systems und den Test des installierten Systems, auch Abnahmetest genannt, untergliedern. Im Funktions- und Modultest werden, wie der Name schon sagt, die einzelnen Funktionen und Module auf ihre Spezifikation hin getestet. Grundlage dafür sind die Dokumente des Software-Feinentwurfs. Hier sind die Aufgaben der Funktionen und Module genau formuliert, so daß aus diesen die Testfälle abgeleitet werden können. Ziel dieser Testphase ist es zu prüfen, inwieweit das Modul seinen Aufgaben bei der Aufgabenerfüllung im Gesamtsystem gerecht werden kann. Beim Test ist zu bemerken, daß einzelne Funktionen oder Module in einer Beziehung zu anderen Funktionen und Modulen stehen können, die noch nicht fertig implementiert oder für diesen einzelnen Test nicht relevant sind. Diese fehlenden Teile müssen durch entsprechende Werkzeuge beim Test simuliert werden.
Der Subsystemtest rückt Gruppen von Modulen, welche sich in einer mehr oder minder engen funktionalen oder datenflussbezogenen Abhängigkeit befinden, in den Mittelpunkt der Untersuchung. Grundlage ist der Grobentwurf, in welchem die Abhängigkeiten der Subsysteme untereinander und ihre Funktionen im Gesamtsystem geschildert sind. Die Gesamtheit aller Subsysteme und eventuell die zur Erfüllung der eigentlichen Aufgabe nötigen Hardwarekomponenten bilden das Sy-

stem. Da hier Hard- und Software zusammengeführt wird, spricht man auch von einem integrierten System.
Die Anforderungen an das Gesamtsystem sind im Systementwurf definiert, wobei natürlich auch zur Testfallermittlung die Anforderungsdefinition hinzugezogen werden kann. Dieser Test sollte, wenn möglich, unter den realen Einsatzbedingungen stattfinden, andernfalls muß eine entsprechende Simulation durchgeführt werden. Am Schluß steht der Abnahmetest. In dieser letzten Phase wird das System auf jeden Fall in seiner realen Umgebung ausgeführt. Hier wird das System gegen die Erfordernisse und Anforderungen des Auftraggebers getestet. Die Tester sind hier meist die Personen, die auch das Produkt nutzen werden. Diese sind natürlich bemüht, Schwachstellen im System zu finden, womit sich die Definition von Myers ([7]) wieder aufgreifen läßt, der Testen damit begründet, Fehler zu finden.

3.7 Teststrategien

Wie schon gezeigt wurde, gibt es eine unendliche Anzahl an Tests, die man zu Verifikation und Validation heranziehen kann. Es gibt aber Tests welche mit höherer Wahrscheinlichkeit einen Fehler finden als andere Tests. Teststrategien sind Vorgehensweisen, Tests mit hoher Fehlerwahrscheinlichkeit zu finden. Im Hinblick auf Einschränkung der Testressourcen durch die Planung, werden unterschiedliche Strategien angewandt. Diese haben alle ihre Vor- bzw. Nachteile. Durch methodisches Vorgehen vergewissert man sich, daß die ausgewählten Tests mit hoher Wahrscheinlichkeit einen Fehler aufgedeckt hätten oder ihn aufdecken würden, wenn die Software nicht korrekt implementiert wurde. Auch die Anzahl solcher “starker“Tests kann erdrückend sein, so daß man zu einer Auswahl hierbei gezwungen wird. Vollständige Sicherheit kann nie über richtige Auswahl an Tests erlangt werden. Dieses Problem wird auch als “test case selection“Problem bezeichnet.
Um dieses Problem eindämmen zu können werden zunächst die idealen Fehlerbedingungen erarbeitet. Die dadurch gewonnenen Erkenntnisse werden verwendet, um eine Verkleinerung der Fehlerquellen per Systemskalierung vorzustellen. Daraufhin werden einige Teststrategien zur Bestimmung von starken und weniger starken Tests erläutert.

3.7.1 Ideale Fehlerbedingungen

Nach Definition ist ein Testobjekt fehlerbehaftet, wenn sich unter einer gegebenen Eingabe eine Abweichung der Ausgabe von der erwarteten Ausgabe beobachten läßt. Ein Fehler muß also durch eine Eingabe erzeugt werden und sein Auftreten muß eine

Abweichung vom erwarteten Ausgabewert erzeugen. Hieraus ergeben sich die idealen Fehlerbedingunen (Ideal fault conditions):

Erreichbarkeit (reachability) Die fehlerhafte Anweisung muß ausgeführt werden, um einen Fehler zu provozieren.

Notwendigkeit (necessity) Eine fehlerhafte Anweisung muß eine Abweichung vom korrekten Ergebnis ergeben. Wenn z.B. die Anweisung x=y/2 anstelle der richtigen Zuweisung x=y implementiert wurde, reicht es nicht, die fehlerhafte Anweisung mit dem Wert 0 für y auszuführen.

Propagierung (propagation) Die Abweichung muß in den Ausgaben des Testobjekts beobachtet werden können.

Diese Regeln sind bei großen Softwarekomponenten schwer einzuhalten. Die Erreichbarkeitsregel macht daher die Wichtigkeit von kontrollflußorientierten Strategien und Testüberdeckungsmessung sehr deutlich. Der interne sowie externe Zustand des Testobjekts wird durch die Notwendigkeitsregel mit einbezogen. Hierzu existieren die datenflußorientierten Teststrategien. Besonders bei Testobjekten mit spärlichen Ausgaben wird die Propagierung erschwert.

3.7.2 Systemskalierung

Teststrategien sind immer nur so gut, wie es die Komplexität des Softwaresystems zuläßt. Ist ein System zu groß oder zu komplex, so kann ein Fehler eventuell gar nicht nach außen propagieren. Dies bedeutet, daß die ausschließliche Verwendung von Systemtests (Tests des gesamten Softwaresystems) selten vorteilhaft sind. Hier sollen wesentliche Nachteile des sogenannten “nicht inkrementiellen“Testens genannt werden und danach auf unterschiedlich skalierte Tests und Methoden des “inkrementiellen“Testens eingegangen werden. Abschließend werden Abnahmetests als Kriterium für auslieferungsfähige Software vorgestellt ([37]).

Inkrementielles versus nicht inkrementielles Testen

Das Testen ohne Zusammensetzen der Komponenten bezeichnet man als nichtinkrementielles Testen. Im sogenannten “big bang“Testen wird nur das System als gesamtes getestet. Das nichtinkrementielle Testen, und sein Spezialfall “big bang“Testen, ist in seiner Konstruktion wenig aufwendig. Die eingesparte Zeit geht aber bei der Fehlersuche während der Systemtests verloren. Besonders bei großen Projekten wird

die Fehleraufdeckung und Fehlersuche durch die in "Ideale Fehlerbedingungen"(siehe Abs. 3.7.1) genannten Voraussetzungen erschwert. Eine fehlerhafte Anweisung kann sich sehr leicht hinter den Transaktionsflüssen der Software verstecken, ohne daß sie von irgendeiner Eingabe an das Programm zur Erscheinung gebracht wird. Von nichtinkrementiellem Testen wird in der einschlägigen Literatur abgeraten.
Üblicherweise wird daher zuerst im kleinen Maßstab getestet, um die Übersichtlichkeit zu wahren und potentielle Fehlerquellen einzugrenzen. Wie der Maßstab gewählt wird, obliegt dem Testentwickler. Nachdem eine ausreichende Anzahl an Komponenten getestet ist, wird der Maßstab vergrößert und neue Tests erstellt sowie getestet. Auf diese Art und Weise werden die einzeln getesteten Komponenten Stück für Stück in das gesamte System integriert. Dieses Verfahren nennt sich inkrementielles Testen ([10] S. 44). Am Schluß wird das komplette Softwaresystem getestet.
Durch die Aufteilung des Systems in kleinere, aber besser testbare Einheiten oder Komponenten hat man zwar einen höheren Konstruktionsaufwand mit Vertreterfunktionalitäten (siehe 6.3.2), aber bessere Kontrolle über den Testprozeß.

Unit-, Integration- und Systemtests

Beim inkrementiellem Testen zerlegt man das Softwaresystem zunächst in kleine, in ihrer Funktionalität meist abgeschlossene Einheiten. Man unterscheidet je nach Größe des ausgewählten Ausschnitts in:

Unittest Dies ist ein Test mit der kleinsten Einheit der Testskala.

Integrationstest Hier werden mehrere Units gemeinsam geprüft, um die Schnittstellen der Komponenten zu testen.

Systemtest Im Systemtest wird dann das ganze Programm oder Softwaresystem geprüft.

Ein **Modultest** bezieht sich auf ein Modul des Systems, es ist eine Bezeichnung für Unittest. Ein Interfacetest bezieht sich wie ein Integrationstest auf ein oder mehrere Schnittstellen eines Subsystems.

Stubs, Mocks und Treiber

Die aufgeteilten Einheiten folgen meist einer Benutzungshierarchie. Sind die Einheiten zum Beispiel Packages ist es eine Aufrufhierarchie. Bei Unittests und Integrationstests müssen die nicht vorhandenen Einheiten ausgeblendet werden, jedoch so,

daß das Testobjekt lauffähig bleibt. Je nachdem welche Einheiten zuerst getestet werden, müssen Treiber, Stubs oder beides für die ausgeblendeten Einheiten erstellt werden.
Im Top-Down Testen werden die Einheiten von unten nach oben zuerst getestet, wobei die benutzten Einheiten durch Vertreter (Mocks oder auch Stubs genannt) ersetzt werden.
Im Bottom-Up Testen werden die Einheiten von unten nach oben zuerst getestet, wobei ihre Benutzung durch sogenannte Treiber simuliert wird.
Im Sandwichtesten werden Einheiten an beliebiger Stelle der Hierarchie unter Zuhilfenahme von einem Treiber und Stubs getestet.
Jede Strategie hat Vor- und Nachteile ([8] S. 100). Die Wahl der Strategie ist beliebig und kann mit anderen kombiniert werden. In der Regel wird man diejenigen Einheiten testen wollen, die zuerst in implementierter Form vorliegen ([10] S. 46).

Abnahmetests

Am Ende des Entwicklungsprozesses, nach den Systemtests, werden Abnahmetests (acceptance tests) durchgeführt. Die Abnahmetests zeigen aus der Sicht des Anwenders, daß das System ein Mindestmaß dessen bietet, was er an Qualität (Performance, Benutzbarkeit, etc.) fordert. Meist sind die Kriterien für den Abnahmetest schon bei der Ausarbeitung der Anforderungsdokumente festgelegt und gegebenenfalls vertraglich abgesichert worden. Die Planung der Abnahmetests liegt also als erstes vor, und die Abnahmetests werden als letzte Tests getestet. Die zeitliche Anordnung von Planung und Ausführung von Unittests, Integrationstests und Abnahmetests wird in dem V-Modell sehr gut illustriert (Abb. 3.2). Weitere Informationen hierzu finden sich unter ([13] S.20).

3.8 Testverfahren

Ein Testverfahren bezeichnet eine begründete Vorgehensweise (in der Regel in Form von festgelegten Teilaufgaben) zum Erreichen bestimmter Ziele, wie z.B. zur Aufdeckung einer bestimmten Klasse von Fehlern.
Es gibt sehr unterschiedliche Testverfahren. Da Testobjekte in der Regel nicht vollständig sind, daß heißt mit allen denkbaren Testdatenkombinationen getestet werden können, geben Testverfahren Hinweise zur Auswahl von Testfällen und von Testdaten bzw. Testkombinationen. Die verschiedenen Testverfahren unterscheiden sich im wesentlichen dadurch, welche Schwerpunkte bei der Auswahl von Testfällen und

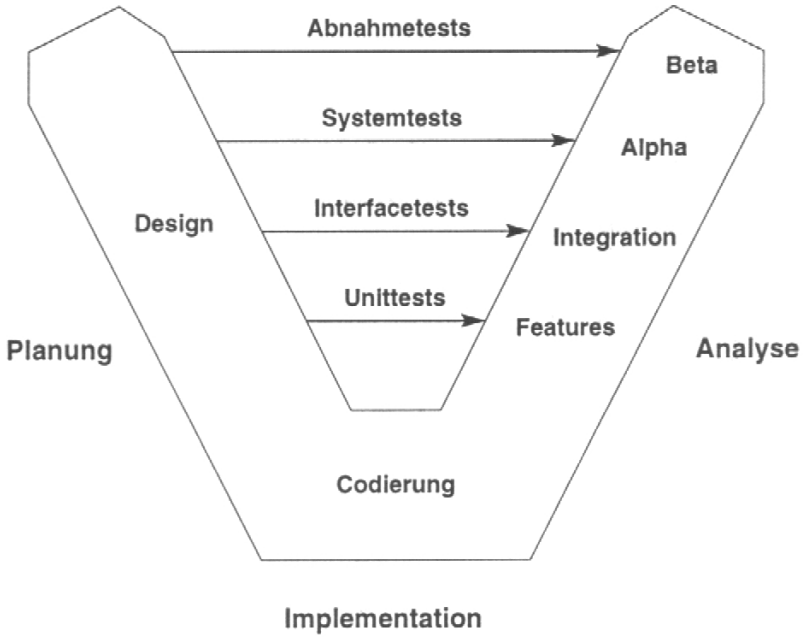

Abbildung 3.2: Das V-Modell zur Planung und Durchführung der Testaktivitäten

Testdaten (-kombinationen) gesetzt werden. Durch diese unterschiedlichen Schwerpunkte ergeben sich auch unterschiedliche Stärken und Schwächen der einzelnen Verfahren. Die Testverfahren können in

- funktionale bzw. funktionsorientierte und
- strukturelle bzw. strukturorientierte Testverfahren

unterteilt werden.
Funktionale bzw. funktionsorientierte Testverfahren benutzen die *Spezifikation*, strukturelle bzw. strukturorientierte Testverfahren benutzen die *Implementierung* des Testobjekts als Referenz für die Bildung von Testfällen.
Die Testverfahren können in dieser Ausarbeitung nur sehr oberflächlich behandelt werden. Zur Vertiefung der Testverfahren werden an geeigneten Stellen Literaturhinweise gegeben.

3.8.1 Funktionale Testverfahren

Funktionale Testverfahren benutzen die *Spezifikation* des Testobjekts (z.B. in Form des Anforderungsdokuments) als Referenz für die Bildung von Testfällen. Beispiele für die funktionale Testfallermittlung sind die intuitive Testfallermittlung, die Funktionsabdeckung, die Äquivalenzklassen-Analyse, die Grenzwertanalyse sowie die Ursache-Wirkungs-Analyse.
Diese Verfahren werden in den nachfolgenden Abschnitten kurz vorgestellt.

Intuitive Testfallermittlung

Bei der intuitiven Testfallermittlung werden Testfälle intuitiv, auf der Basis von Erfahrungswerten, erzeugt.

Funktionsabdeckung

Das Testverfahren der Funktionsabdeckung ist auf das Normalverhalten des Testobjekts ausgerichtet (Testfälle werden anhand der spezifizierten Funktionen gebildet). Hierbei wird für jede Funktion eine Eingabe- sowie Ausgabespezifikation erstellt. Auf Basis dieser Eingabe- sowie Ausgabespezifikation werden die Testdaten generiert. Die Funktionsabdeckung ist in der Regel Bestandteil anderer Testverfahren.

Äquivalenzklassenanalyse

Eine Äquivalenzklasse ist eine Klasse von Eingabewerten, die ein identisches funktionales Verhalten verursacht. Von einer Äquivalenzklasse nimmt man an, daß der Test mit einem beliebigen Wert aus dieser Klasse äquivalent ist zu dem Test jedes anderen Wertes dieser Klasse:

- Wenn der Wert einen Fehler aufdeckt, erwartet man, daß auch jeder andere Wert der Äquivalenzklasse diesen Fehler aufdeckt.
- Wenn der Wert keinen Fehler aufdeckt, erwartet man, daß auch kein anderer Wert der Äquivalenzklasse einen Fehler aufdeckt.

Grenzwertanalyse

Nachdem die Eingabebereiche identifiziert wurden, werden Werte in der Nähe der Grenzen der Bereiche ausgewählt. Es werden Tests mit Werten knapp innerhalb der Grenzen, auf den Grenzen und knapp außerhalb der Grenzen gewählt. Testdaten, die Grenzwerte abdecken, haben meist eine höhere Wahrscheinlichkeit Fehler aufzudecken, als Testdaten die dies nicht tun. Eine umfaßende Definition sowie Erläuterungen zur Grenzwertanalyse finden sich unter ([10] S. 132).

Ursache- Wirkungs -Analyse

Eine Schwäche der vorhergehenden Testmethoden ist, daß sie keine Kombination von Werten testen. Die Ursache-Wirkungs-Analyse ([11] S. 156) berücksichtigt Wirkungen, die von einer Kombination von Ursachen, zumeist Eingaben, erzeugt werden. Hierzu wird die vorliegende Spezifikation in einen logischen Graphen übersetzt. Die Spezifikation kann auch natürlichsprachlich sein. Anhand des Graphen werden alle Ursachenkombinationen, die eine Wirkung erzielen, in eine Entscheidungstabelle aufgetragen. Eine Kombination die mehrere Wirkungen erzielt, wird für jede weitere Wirkung nochmals eingetragen. Aus der Entscheidungstabelle werden dann die Testfälle abgeleitet. Die Ursache-Wirkungs-Analyse ist gut dazu geeignet Unvollständigkeiten in der Spezifikation aufzudecken.

3.8.2 Strukturelle Testverfahren

Strukturelle Testverfahren benutzen die Implementierung des Testobjekts als Referenz für die Bildung von Testfällen. Strukturelle Testverfahren lassen sich unterteilen in kontrollflußorientierte Testverfahren und datenflußorientierte Testverfahren. Bei den kontrollflußorientierten Testverfahren werden Strukturelemente (z.B. Anweisungen) zur Erzeugung von Testfällen verwendet. Bei den datenflußorientierten Testverfahren werden Zugriffe auf Variablen (z.B. Definitionen) zur Erzeugung von Testfällen verwendet.

Kontrollflußorientierte Testverfahren

Bei den kontrollflußorientierten Testverfahren werden Strukturelemente (z.B. Anweisungen, Zweige, Bedingungen) zur Erzeugung von Testfällen verwendet. Es gibt unterschiedliche kontrollflußorientierte Testverfahren. Dabei werden Testabdeckungskenngrößen (Cx) als Testziele verwendet. Vorgaben für Testziele werden in der Form

“n% Cx“spezifiziert ([11] S. 62-108). Das Ziel “100% C1“bedeutet z.B. daß die Testdaten so gewählt werden müssen, sodaß jeder Zweig eines bestimmten Programmcodes mindestens einmal durchlaufen wird. Entsprechende Messungen sind praktisch nur mit Werkzeugunterstützung durchführbar.
Im folgenden werden die wichtigsten kontrollflußorientierten Testverfahren, bzw. die diesen Verfahren zugrunde liegenden Kriterien zur Auswahl von Testfällen kurz aufgelistet:

C0 Anweisungsüberdeckung Verhältnis von Anzahl der mit Testdaten durchlaufenen Anweisungen zur Gesamtanzahl der Anweisungen.

C1 Zweig- / Entscheidungsüberdeckung Verhältnis von Anzahl der mit Testdaten durchlaufenen Zweige zur Gesamtanzahl der Zweige.

C2 Bedingungsabdeckung Verhältnis von Anzahl der mit Testdaten durchlaufenen Prädikate (Terme innerhalb von Entscheidungen) zur Gesamtanzahl der Prädikate.

C3 Abdeckung aller Bedingungskombinationen Verhältnis von Anzahl der mit Testdaten durchlaufenen Bedingungskombinationen zur Gesamtanzahl der Bedingungskombinationen.

C4 Pfadabdeckung Verhältnis von Anzahl der mit Testdaten durchlaufenen Pfade zur Gesamtanzahl der Pfade.

Datenflußorientierte Testverfahren

Bei den datenflußorientierten Testverfahren werden Zugriffe auf Variablen (z.B. Definitionen) zur Erzeugung von Testfällen verwendet. Zuerst werden bei diesen Testverfahren die Testdaten auf der Basis von Variablenzugriffen ausgewählt. Danach werden die Variablen durch das Programm verfolgt. Hierbei werden kritische Verknüpfungen zwischen Definition und Benutzung von Variablen getestet. Die Testverfahren helfen somit Pfade zu selektieren, die bestimmte Sequenzen von Ereignissen im Zusammenhang mit Daten bzw. Variablen abdecken. Typische Beispiele für potentielle Fehlerkategorien sind die Verwendung von undefinierten Variablen, definierten aber nicht verwendeten Variablen und solchen die mehrfach definiert werden, ohne zwischenzeitlich verwendet zu werden ([11] S. 109-148).

3.9 Verläßlichkeitstesten

Der Begriff Verläßlichkeitstesten (Reliability Test) beinhaltet empirische Testmethoden. Sie basieren auf funktionalen und strukturellen Testmethoden, haben aber wegen ihres statistischen Charakters eine gesonderte Stellung. Obwohl auch beim Reinraum-Prozeß (Cleanroom Process) Wert auf exakte Auslegung der Spezifikation gelegt wird, liegt der eigentliche Ansatz aller statistischen Methoden in der ingenieursmäßigen Verfahrensweise.

3.9.1 Leistungstests

Beim Leistungstest (Performance Test) sollen die Leistungsdaten des Softwaresystems ermittelt werden. Dies wird durch Messen von Eckdaten während des Tests, dem sogenannten benchmarking, erreicht. Die Ermittlung dieser Daten kann sehr kostspielig sein. Die Eckdaten werden in einem Systemprofil aufgetragen. Die Ergebnisse der Leistungstests werden als Ausgangsbasis für Lasttests und Streßtests benutzt ([37]).

3.9.2 Lasttests

Mit Lasttests (Load Tests oder auch Volume Tests genannt) wird das Softwaresystem aus Sicht des Benutzers getestet. Die Eingaben werden so gewählt, daß das System immer höheren Belastungen ausgesetzt ist. Dies kann der Verbrauch von Rechenzeit oder von Ressourcen (z.B. Speicher, Netzkapazität, etc.) sein. Ziel ist es, die Grenzen des Systems aufzudecken, innerhalb derer es zuverlässig operiert. In Lasttests werden die Ressourcen über eine sehr lange Zeit hinweg erschöpft (z.B. mehrere Stunden oder Tage). Die zeitbezogene Aufzeichnung relevanter Eckdaten für Ressourcen ist die wichtigste Aktivität bei dieser Art von Test ([37]).

3.9.3 Streßtests

Bei Streßtests werden die Eingaben so gewählt, daß das System extremen Verhältnissen ausgesetzt wird. Dies sind Eingaben, wie sie unter normalen Umständen sehr unwahrscheinlich sind, z.B. extrem fehlerhafte Eingaben. Man versucht hierdurch zu prüfen, ob das System eine große Variation, bezüglich der angenommenen Durchschnittswerte, verkraften kann. Streßtests benötigen im allgemeinen nicht so lange wie Lasttests. Jedoch ist auch hier die Aufzeichnung der relevanten Eckdaten wichtig ([37]).

3.9.4 Reinraumtests

Diese statistische Methode ist Teil des Reinraum-Prozesses (Cleanroom Process). Bei herkömmlichen Modellen wird das erst teilweise entwickelte System bereits Softwaretests unterzogen. Häufig ist es jedoch so, daß die fein strukturierten Elemente erst am Ende des Entwicklungsprozesses implementiert werden. Da die volle Funktionalität erst am Ende des Entwicklungsprozesses vorliegt, müssen die Tests mehrfach angewendet und gegebenenfalls angepaßt bzw. verfeinert werden. Es gibt konventionelle und gemäßigte Ansätze einen Reinraum-Prozeß durchzuführen ([12]).

3.9.5 Regressionstests

Mit jeder Veränderung der Software, z.B. weil eine Fehlerkorrektur vorgenommen werden mußte, wird eine erneute Validierung des Systems erforderlich. Dies erreicht man entweder durch Anpassung der Tests und erneutes Testen derselbigen oder aber durch Neuentwicklung von Tests. Hierdurch will man sicherstellen, daß die Veränderungen keinen Rückschritt, eine Regression, der Software bewirkt haben. Der Prozeß des Testens auf Rückschritt nennt sich Regressionstesten. Inhalt eines jeden Regressionstests ist der Vergleich auf Erhaltung der Funktionalität der Software.
Die unterschiedlichen Typen von Ausgaben erschweren es, eine allgemeine oder formale Strategie zu verfolgen. Das Regressionstesten einer graphischen Benutzerschnittstelle auch GUI Testen genannt, erfordert z.B. capture/replay Werkzeuge. Die Automatisierung des Regressionstestens insgesamt wird aber als sehr wichtig eingestuft. Es ist auf diese Weise möglich, Regressiontests in eine Testsuite zu gruppieren. Diese Testsuite kann bei jeder Änderung erneut gestartet werden und teilt dem Entwickler automatisch mit, wenn die Änderung der Implementierung nicht validiert wurde.
Eine automatische Regressionstestumgebung besteht aus einem sogenannten Test Harness oder Testtreiber und einer Testsuite, die mit dem Testtreiber gestartet werden kann. Der Testtreiber ist für Konstruktion, Ausführung und Rücksetzung jedes Tests zuständig. Die Konstruktion besteht in der Koordination von Dateien. Daraufhin wird das für den Test repräsentative Programm gestartet bzw. ausgeführt. Die Rücksetzung besteht in der Entfernung überflüssiger Objekte, wodurch der Initialzustand des Tests wieder hergestellt wird. Ausführungszeiten und Testergebnisse werden in ein sogenanntes “journal file“protokolliert ([13] S. 55).

3.10 Review Meetings

Eine weitere Möglichkeit Software zu validieren besteht im formellen Testen, den sogenannten Review Meetings. Diese werden von einer Personengruppe durchgeführt. Ziel ist es hierbei, sich einen Überblick zu einem Aspekt der Software zu verschaffen. Hierdurch werden fehlerhafte Handlungen im Entwicklungsprozeß, aber auch ungelöste Problemstellungen aufgedeckt.
Ausgewählte, am Softwareprozeß beteiligte Personen treffen sich, um über das Design zu befinden. Dies schließt einen der Entwickler des Designs mit ein. Die beteiligten Personen werden in ausreichender Zeit vor dem Meeting mit den notwendigen Desgindokumenten versehen. Während des Meetings werden Probleme und Lösungen hierzu angesprochen und aufgezeichnet, jedoch nicht ausformuliert. Die maximale Dauer eines solchen Meetings sollte nicht mehr als zwei Stunden betragen. Nach dem Meeting verfaßt der Schriftführer einen Bericht, der nun zu weiteren Aktivitäten herangezogen werden kann, wie z.B. zur Abnahme des Designs.
Man unterscheidet in Inspections, Technical Reviews und Walkthroughs. Wird ein Review Meeting programmbezogen durchgeführt, spricht man von Code Inspection, Code Review oder Code Walkthrough.
Review Meetings sind sehr effektiv in Bezug auf die Aufdeckung von Fehlhandlungen und können hohen Testaufwand vermeiden. Weiterführende Literatur ist
([10] S. 39) sowie ([8]).

Kapitel 4

Testen objektorientierter Software

In diesem Kapitel wird das Testen von objektorientierter Software vorgestellt. Hierbei werden die Besonderheiten der Objektorientiertheit sowie ein Komplexitätsvergleich von objektorientiertem gegenüber prozeduralem Testen betrachtet. Desweiteren wird der Begriff des Testens unter Einsatz der Reflection Technik beschrieben und dem manuellen Testen gegenübergestellt. Nachfolgend wird das Extreme Programming Paradigma (XP) und dessen Test First Ansatz vorgestellt. Das Kapitel wird abgeschlossen durch eine Bewertung der Anwendbarkeit traditioneller Testmethodik bezüglich objektorientiertem Testen und durch eine Erläuterung der hierfür vorhandenen Teststufen.

4.1 Besonderheiten der Objektorientierung

Im folgenden Unterabschnitt wird auf die Besonderheiten der objektorientierten Programmiersprachen eingegangen. Die drei Hauptbesonderheiten der Objektorientierung, im Gegensatz zu den prozeduralen Programmiersprachen sind: die Kapselung, die Vererbung und die Polymorphie.

4.1.1 Kapselung

Die Kapselung ist kein völlig neues Konzept der Programmiersprachen. Jedoch ist sie sehr viel stärker ausgeprägt als bei der prozeduralen Programmierung. So sollte der Zugriff auf die Daten eines Objektes nach Möglichkeit nur über Methoden des Objekts erfolgen und Daten, die nur für die interne Implementierung der Klasse von Bedeutung sind gar nicht von außen zugänglich sein. Diese Daten bestimmen jedoch den Zustand des Objektes, welcher beispielsweise vor jedem Test definiert, gesetzt

und nach jedem Test kontrolliert werden sollte. Meist bieten objektorientierte Programmiersprachen jedoch Konzepte, diese Kapselung zu durchbrechen und somit beispielsweise den Testtreibern vollständigen Zugriff auf die Daten eines Objektes zu gewähren (z.B. Friend Klassen in C++).
Die Kapselung in Java bringt aber auch Vorteile. Dadurch, daß die Objekte nach außen abgeschlossen sind und ein Zugriff nur über fest definierte Schnittstellen möglich ist, wird beispielsweise der Test einzelner Objekte unabhängig voneinander erleichtert, und Fehlerursachen lassen sich leichter lokalisieren.

4.1.2 Vererbung

Die Vererbung gewährt abgeleiteten Klassen direkten Zugriff auf Elemente der Basisklasse, lockert damit das Kapselungsprinzip, wodurch leichter unerwünschte Seiteneffekte und Fehler entstehen können. Die Methoden der Basisklasse werden außerdem in der Unterklasse in einem anderen, verändertem Kontext ausgeführt, indem sie eventuell nicht mehr fehlerfrei sind. Hieraus folgt, daß abgeleitete Klassen nicht unabhängig von ihren Basisklassen getestet werden können.
Vor allem bei tiefen Verbindungshierarchien geht der Überblick über alle geerbten Methoden und Attribute leicht verloren, wodurch es leicht zu unbeabsichtigter Wiederverwendung bereits vorhandener Namen und dem Überschreiben der geerbten Elemente kommen kann. Geerbte Methoden sind oftmals nicht sinnvoll und müssen angepaßt werden, was leicht vergessen werden kann (z.B. copy() oder isEqual()). Die Möglichkeit der Mehrfachvererbung enthält einige weitere Fehlerquellen, ist jedoch nicht in allen objektorientierten Programmiersprachen möglich.

Abstrakte Klassen und Schnittstellen

Über den Sinn des Tests abstrakter Klassen oder Schnittstellen läßt sich streiten. Will man sie jedoch testen, gibt es zwei Möglichkeiten dies zu tun. Zum einen kann man die konkreten Ableitungen der Klassen testen, wobei man eventuell die Testklassen von einer parallelen Testhierarchie ableitet. Zum anderen kann man eine konkrete Unterklasse extra für den Test erzeugen. Die Komplexität einer abstrakten Klasse ist meist sehr gering, so daß sich der Testaufwand dafür nicht rechtfertigt, es sei denn, es existiert gar keine konkrete Ableitung, was vor allem bei der Entwicklung von Frameworks auftritt.

4.1.3 Polymorphie

Das Hauptproblem welches die Polymorphie mit sich bringt, besteht in der dynamischen Bindung von Methoden. Dynamische Bindung bedeutet, daß erst zur Laufzeit entschieden wird, welche Funktionen in welchem Objekt einen Auftrag erledigen. Dadurch ist der Programmablauf nicht mehr statisch aus dem Programmtext ableitbar, d.h. bei der Entwicklung der Tests ist unter Umständen gar nicht genau klar, welche Bindungen zur Laufzeit auftreten können.
Ein kleines Beispiel (siehe Abb. 4.1) soll dies verdeutlichen: Es gibt eine Klasse Vieleck und eine daraus abgeleitete Klasse Rechteck. Die Klasse Vieleck definiert eine Methode getUmfang(), welche in der Klasse Rechteck überschrieben wird. Der Programmausschnitt (siehe Abb. 4.1) zeigt wie erst zur Laufzeit entschieden wird, welche Methode Verwendung findet. Daher müssen alle möglichen dynamischen Ab-

```
...
Vieleck a;
Vieleck b;
...
a = new Vieleck();
b = new Rechteck();
...
if (BedingungDieZurLaufzeitErmitteltWurde)
{a = b;}
System.out.println(a.getUmfang());
...
```

Abbildung 4.1: Ein Beispiel zum dynamischen Binden

lauffolgen getestet werden. Durch mehrfache Wiederholung der Polymorphie kann die Anzahl möglicher Ablaufpfade geradezu explodieren. Dadurch gestaltet sich die vollständige Abdeckung oft als sehr schwierig.

4.2 Komplexitätsvergleich gegenüber prozeduralem Testen

Lange Zeit beschäftigte man sich entweder gar nicht mit dem Test objektorientierter Software oder ging ohne genauere Untersuchung davon aus, daß man bisher bekannte Prüfverfahren unverändert übernehmen könnte. So schrieb Grady Booch noch 1994:

> "... the use of object-oriented design doesn't change any basic testing principles; what does change is the granularity of the units tested."
> ([14])

James Rumbaugh behauptete anfangs der Neunziger sogar, der Aufwand für den Test würde sich reduzieren:

> "Both testing and mainenance are simplified by an object-oriented approach..."([15])

Es gab jedoch auch andere Stimmen. So schrieb Boris Bezier 1994:

> "... it costs a lot more to test objectoriented-software then to test ordinary software - perhaps four or five times as much ... Inheritance, dynamic binding and polymorphism creates testing problems that might exact a testing cost so high that it obviates the advantages."([16])

Der Wahrheit kommt Bezier wahrscheinlich näher. Der Test objektorientierter Software ist nicht dasselbe oder um ein vielfaches aufwendiger als der Test traditioneller (prozeduraler) Software. Das Programmieren und Testen in einer objektorientierten Programmiersprache ist unterschiedlich hinsichtlich der Verwendung einer prozeduralen Programmiersprache. Objektorientiertes Programmieren hat neue Konzepte die mit Sicherheit einige Vorteile beim Design und der Implementierung bringen, die aber auch neue Fehlerquellen enthalten, welche es früher nicht gab. Die speziellen Eigenheiten und Fehlerquellen gilt es beim Testen zu berücksichtigen.

4.3 Reflection Testen versus manuelles Testen

Das Reflection-Modell erlaubt es, Klassen und Objekte, die zur Laufzeit im Speicher gehalten werden, zu untersuchen und in begrenztem Umfang zu modifizieren. Das Konzept der Reflection (oder auch Introspektion) wird dann besonders interessant, wenn wir uns mit Hilfsprogrammen zum debuggen beschäftigen oder GUI-Builder schreiben. Diese Programme nennen sich auch Meta-Programme, da sie auf den Klassen und Objekten anderer Programme operieren. Reflection fällt daher auch in die Schlagwortkategorie Meta-Programming.
Im Gegensatz zum manuellen Testen, in dem die Tests welche ausgeführt werden sollen, vom Tester explizit angegeben werden, wird beim reflection Testen jeder Testfall herangezogen. Ein weiterer Unterschied besteht in der Ausführungsreihenfolge. Während beim manuellen Testen die Ausführungsreihenfolge immer fix vorgegeben ist, wird beim reflection Testen die Ausführungsreihenfolge der Tests, wegen dem dynamischen Binden der Klassen zur Testsuite variieren. Ein Vorteil des reflection Testens ist, daß bei neu hinzugefügten Testfällen die Testsuite nicht ergänzt

werden muß, da sie gar nicht verwendet wird. Somit werden auch keine Testfälle beim Testlauf vergessen. Desweiteren werden durch reflection Tests eventuelle Beeinträchtigungen sowie Interferenzen zwischen einzelnen Testfällen eher aufgedeckt. Ein Vorteil vom manuellen Testen ist, daß man bei einem Testlauf nur bestimmte Tests laufen lassen kann und nicht immer die Gesamtheit aller Testfälle durchlaufen werden muß, was bei großen Projekten mit vielen Testfällen zu einer signifikanten Zeitersparnis führen kann.

4.4 Extreme Programming

Extreme Programming (XP) gehört zur Gruppe der agilen Softwareprozesse. Hierbei handelt es sich um ein Prozeßmodell zur objektorientierten Softwareentwicklung. Es eignet sich vor allem für kleinere Projekte mit unklaren und sich ändernden Anforderungen. Dabei werden sowohl an die Entwickler als auch an den Auftraggeber (Kunden) hohe Anforderungen gestellt. Extreme Programming basiert auf vier Grundwerten: Kommunikation, Feedback, Einfachheit und Mut. Dazu kommen etwa ein Dutzend Regeln und Praktiken, welche hier auszugsweise aufgeführt werden:

Kunde vor Ort Um eine möglichst einfache und direkte Kommunikation mit dem Kunden zu ermöglichen, sollte immer ein Endanwender ins Projektteam integriert werden. Nach jeder Iteration erhält der Kunde ein lauffähiges Produkt und kann darauf Einfluß nehmen (Anpassung der Anforderungen).

Kleine Versionen Jede neue Version soll so klein wie möglich sein, aber zugleich die für das Gesamtprojekt wichtigsten Erweiterungen enthalten. Kleine Versionen garantieren eine schnelle Rückantwort des Benutzers (Userfeedback) und ein einfacheres Zeitmanagement des Projektes.

Pair Programming Zwei Entwickler arbeiten am selben Terminal. Einer tippt, der andere überprüft, denkt mit oder denkt voraus. Damit werden eine Menge Fehler schon gesehen und korrigiert, bevor das Programm erstmals getestet wird. Auf den ersten Blick kommt man zwar nur halb so schnell vorwärts, da zwei Personen das tun, was sonst einer allein macht. Auf den Gesamtaufwand der Entwicklung inklusive Tests ist man aber wieder gleich schnell, da viele Fehler erst gar nicht entstehen.

Refactoring Neben Tests ist Refactoring (etwas neu herstellen) eines der wichtigsten Merkmale von XP. Mit Refactoring meint man die Anpassung des Designs und der Implementierung ohne Veränderung der Funktionalität. Da bei jeder

Version jeweils nur das nötigste implementiert werden soll, ist die Gefahr groß, daß bei Erweiterungen das Design den Bedürfnissen des Kunden entspricht. Anpassungen sind hier also dauernd nötig.

Eine Gewichtung der einzelnen Regeln und Praktiken ist schwierig. Sie können auch einzeln eingesetzt werden, jedoch entfalten sie laut Kent Beck erst gemeinsam die volle Wirkung. Schlußendlich befolgt man einen iterativen Prozeß, welcher nach einem sehr strengen Muster abläuft:

- Anforderungen aufnehmen: Dies geschieht mit Hilfe von "User Story Cards", wo der Kunde eine bestimmte Funktionalität beschreibt. Die Entwickler legen darauf hin die Zeit fest, in der diese Anforderungen erfüllt werden können. Die Rollen können auch vertauscht werden, das heißt, der Kunde legt den Zeitrahmen fest und der Entwickler gibt an, was in dieser Zeit in einem bestimmten Modul alles für Features realisiert werden können. Durch Gruppieren der User Story Cards werden Umfang und Zeitpunkt der einzelnen Versionen festgelegt. Dieser ganze Vorgang wird auch Planungsspiel genannt.
- Entwicklungsphase: In der Entwicklungsphase durchläuft der Prozeß mehrere Iterationen. Am Ende jeder Iteration steht eine lauffähige Version der Software zur Verfügung. Jede Iteration besteht aus Implementierung (Pair Programming), Refactoring und Tests. Am Ende einer Iteration wird der gesamte Versionsplan überprüft, die Inhalte oder der Zeitrahmen nötigenfalls angepaßt.

Extreme Programming ist nicht in jedem Fall gut geeignet. Große Teams, nicht vertrauenswürdige Kunden und Technologien können den Einsatz von Extreme Programming unmöglich machen. Detaillierte Informationen zu Extreme Programming sind im Buch von Kent Beck ([17]) oder auf zahlreichen Internetseiten zu finden.

Der Test- First Ansatz

Test-First (Teste zuerst) ist eine Vorgehensweise bei der Koordinierung von Softwaresystemen. Test-First ist nicht nur eine reine qualitätssichernde Tätigkeit, sondern steuert auch das Softwaredesign in Richtung Testbarkeit und Einfachheit. Folgende Punkte beschreiben das ideale Test-First-Vorgehen:

- Bevor man eine Zeile Produktionscode schreibt, entsteht ein entsprechender Test, der diesen Code motiviert.

- Es wird nur so viel Produktionscode geschrieben, wie es der Test verlangt.
- Die Entwicklung findet in kleinen Schritten statt, in denen sich Testen und Kodieren abwechseln. Eine solche "Mikro-Iteration"dauert nicht länger als 10 Minuten.
- Zum Zeitpunkt der Integration von Produktionscode ins Gesamtsystem müssen alle Unittests erfolgreich laufen.

Dieses kleine Regelwerk mag dem einen oder anderen Programmierer als willkürlich erscheinen und ihrer persönlichen Erfahrung widersprechen. Einige Vorteile liegen jedoch auf der Hand:

- Jedes einzelne Stück Code ist getestet. Dadurch werden Änderungen, die vorhandene Funktionalität zerstören, sofort entdeckt.
- Die Tests dokumentieren den Code, da sie im Idealfall sowohl die normale Verwendung als auch die erwartete Reaktion in Fehlerfällen zeigen.
- Die Kürze der Mikro-Iterationen führt zu einem äußerst schnellen Feedback. In maximal zehn Minuten kann man nur wenig programmieren und daher auch nur wenig falsch machen.
- Das Design des Programms wird maßgeblich von den Tests bestimmt. Dies führt fast immer zu einem einfacheren Design, als wenn es am Reißbrett entworfen worden wäre, da für komplexe Strukturen nur selten einfache Tests geschrieben werden können.

Detaillierte Informationen zum Test-First Ansatz finden sich im Buch von Johannes Link ([18]).

4.5 Anwendbarkeit traditioneller Testmethodik

Ein wichtiger Aspekt des Testens von objektorientiertem Programmcode ist die Auswahl der Testmethodik. Im folgenden Abschnitt werden die traditionellen Testmethodiken, welche auch noch beim Testen von objektorientierten Programmen verwendet werden können, vorgestellt.

4.5.1 Reviews Meetings

Reviews Meetings oder Inspektionen für die Prüfung von objektorientiertem Programmcode gestalten sich sehr viel schwieriger, als bei prozeduralem Programmcode. Der Kontrollfluß eines objektorientierten Systems ist stark verteilt auf viele kleine Methoden und verschiedene Objekte, so daß sich die Komplexität von strukturellen Einheiten in deren Beziehungsgeflecht verlagert. Erschwerend kommt die dynamische Bindung hinzu, was zu einer geringen Übereinstimmung des Programmcodes und dem dynamischen Programmverhalten führt.

4.5.2 Black- Box Testen

Beim Black-Box Test wird das Gesamtsystem gegen seine Spezifikation geprüft. Die interne Implementierung wird dabei nicht betrachtet bzw. ist unbekannt. Daher ist dieser Test auch für objektorientierte Systeme sinnvoll. Der Test verläuft genauso wie bei nicht objektorientierten Systemen.

4.5.3 White- Box Testen

Der White-Box Test geht speziell auf die interne Realisierung des Systems ein. Das ist natürlich auch bei objektorientierten Systemen sinnvoll, jedoch muß ein solcher Test die Besonderheiten der Objektorientierung berücksichtigen.
Die Überdeckungskriterien des traditionellen White-Box Tests, die sich auf den Kontrollflußgraphen der Funktionen stützen (Zweigüberdeckung, Bedingungsüberdeckung, Pfadüberdeckung, Anweisungsüberdeckung) besitzen nur eine geringe Aussagekraft, da Methoden in objektorientierten Programmen im allgemeinen sehr klein sind und die Komplexität erst durch das Zusammenspiel der verschiedenen Klassen entsteht. Robert V. Binder ([31]) schlägt folgende Checkliste für den Klassentest vor:

- Jede Methode wird ausgeführt (auch get und set Methoden!).
- Alle Parameter und Rückgabewerte werden mit Äquivalenzklassen- und Grenzwerttests geprüft.
- Jede ausgehende Exception wird ausgelöst und jede hereinkommende Exception wird behandelt.
- Jeder Zustand wird erreicht.

- Jede Methode wird in jedem Zustand ausgeführt.
- Jeder Zustandsübergang eines Zustandsübergangsgraph wird durchgeführt um Zusicherungen zu testen.
- Passende Stress- und Lasttests werden durchgeführt.

4.6 Teststufen

Der Test objektorientierter Systeme läßt sich in vier Stufen einteilen: Unittests, Integrationstests, Systemtests, Abnahmetests bzw. Akzeptanztests.

Unittest (Modultest) Bei der nicht objektorientierten Entwicklung wird in der Regel die Prozedur als kleinste unabhängig testbare Einheit angesehen. In der Objektorientierung wäre es naheliegend die einzelnen Methoden in den Klassen als äquivalent zu betrachten. Methoden können im Normalfall jedoch nicht unabhängig voneinander getestet werden. Da die Methoden einer Klasse durch gemeinsam verwendete Attribute und gegenseitige Benutzung untereinander starke Abhängigkeiten aufweisen, sind sie meist nicht oder nur mit zu hohem Aufwand unabhängig testbar. Daher ist in objektorientierten Systemen die Klasse die kleinste unabhängig testbare Einheit und steht somit im Mittelpunkt des Unittests. Ein weiterer Teil des Unittests ist der Modultest, wobei als Modul eine Klassenhierarchie oder eine kleine Menge eng zusammengehöriger Klassen angesehen wird.

Integrationstest Der Integrationstest testet die Kommunikation und Interaktion zwischen einzelnen Klassen, Modulen und Komponenten. Das Ziel des objektorientierten Integrationstest ist die Überprüfung des korrekten Zusammenwirkens von dienstanbietenden (Server) und dienstnutzenden (Client) Objekten unterschiedlicher Klassen, die nicht in einer Vererbungsbeziehung stehen. Die Integration von abgeleiteten Klassen und ihren Basisklassen ist bereits Aufgabe der Klassentests.

Systemtest Ein Systemtest ist ein Test, in dem das gesamte System auf Funktionalität geprüft wird. Hier gibt es keine Besonderheiten bezüglich der Objektorientierung.

Abnahme-, Akzeptanztest Hier gibt es keine Besonderheiten bezüglich der Objektorientierung.

Mehrere Unit- bzw. Integrationstests welche ein und die selbe Klasse testen, werden zu einem **Klassentest** vereint. Man gruppiert mehrere logisch zusammenhängende Klassentests in einer sogenannten Testsuite oder Testreihe. Mehrere Testsuites werden wiederum zu größeren Testsuites zusammengruppiert.

Kapitel 5

Anwendung auf JLiPSD

In diesem und dem nächsten Kapitel wird dargelegt, wie man ein Testframework für JLiPSD erstellt. Hierbei werden einige der grundlegenden Konzepte der Qualitätssicherung und des Softwaretestens als Mindestanforderung an das Testframework ausgewählt.

In diesem Kapitel wird zunächst der Umfang des Testframeworks für JLiPSD definiert. Nach Vorstellung des JLiPSD Projektes und dessen Entwicklungsumgebung wird eine Orientierung für das Testframework vorgenommen. Das Kapitel wird mit der Untersuchung von verschiedenen Testwerkzeugen abgeschlossen, die zur Verwendung in dem Testframework in Frage kommen.

5.1 Definition des Testframeworks

Das Testframework für JLIPSD wird aus der Entwicklungsumgebung sowie einem darin integrierten Testmeßwerkzeug bestehen.
Bestandteil des Testframeworks soll nicht nur die Entwicklungsumgebung zum Arbeiten, sondern auch Dokumente zur Anleitung und Begleitung des Testprozesses sein. Diese Dokumente werden auch Testdokumente oder Testdokumentation genannt.
Das Testsystem ist die zentrale, ausführbare Komponente des JLIPSD Testframeworks. Es wird zur Durchführung der Testläufe verwendet werden.
Begleitend hierzu wird ein Meßwerkzeug für Überdeckung (Coverage) eine weitere ausführbare Komponente des Testframeworks.

5.2 Einführung in JLiPSD

JLiPSD (Java Library for Parallel Systems Daemon) ist die Java-Umsetzung des LiPSD. Der LiPSD ist Bestandteil von LiPS, Library for Parallel Systems, einem System zur verteilten Berechnung.
Die Hauptproblematik des LiPS Systems ist, daß es bisher nur auf Unix Arbeitsplatzrechnern läuft. Deshalb wurde als Diplomarbeit ([33]) hierzu, von Herrn Andreas Müller, die LiPS-Client Seite, LiPSD genannt, nach Java portiert, um den LiPSD auf allen Architekturen zur Verfügung stellen zu können.
Die Aufgabe des LiPSD bzw. des JLiPSD besteht darin, Prozesse verteilt laufen zu lassen. Hierbei gewährt er fehlertoleranten Zugriff zum verteiltem Speicher auf der LiPS Serverseite. Diesbezüglich sei erwähnt, daß die LiPS Serverseite weiterhin nur unter Unix betrieben werden kann.

5.3 Entwicklungsumgebung von JLiPSD

Die JLiPSD Entwicklungsumgebung ist mit steigender Komplexität begleitend zum Projekt entstanden. Mit ihr soll eventuellen Weiterentwicklern, z.B. bei Portierung des LiPS-Servers, die Arbeit erleichtert werden.
Die Entwicklungsumgebung kombiniert unter anderem den Programmiereditor JEdit, das in JEdit integrierte Ant Build-Werkzeug, den Java Development Kit (JDK) Version 1.4.1, das Log4J ([38]) Kontrollausgaben Werkzeug, das Chainsaw Kontrollausgaben Werkzeug (Chainsaw Logging Tool) ([34]) sowie XML (Extensible Markup Language), da diese Sprache von Ant als Format der Build-Dateien benötigt wird.
Das JLiPSD Testframework ist ein weiterer Bestandteil der Entwicklungsumgebung.

5.3.1 JEdit

JEdit ist ein Text Editor für Programmierer. Dieses Produkt ist der führende in Java programmierte Text Editor, mit mehr als 3 Jahren Entwicklungsgeschichte. JEdit ist Open Source Software (GNU Lizenz) und bietet eine Vielzahl von Möglichkeiten:

Wort-Vervollständigung (Word-Completion) Man tippt den Anfang eines Wortes das im aktuellen Puffer vorhanden ist, d.h. bereits einmal geschrieben wurde und JEdit ergänzt den Rest.

Leichte Bedienbarkeit Im Gegensatz beispielsweise zu Emacs ist JEdit viel leichter zu erlernen. Alle Operationen sind wahlweise per Shortcut, per Fenstermenü, per Kontextmenü oder per Toolbar abrufbar. Einstellungen (Farben, Shortcuts, PlugIns, Dateiformat, usw.) sind alle in leicht verständlichen Menüs konfigurierbar.

XML/HTML Auto Tag Closing Wer viel HTML oder XML Dateien schreibt ist auf Schreibhilfen angewiesen, um nicht die Hälfte der Zeit damit zu verbringen endlos lange Tag-Bezeichner einzutippen. JEdit hilft dabei indem es, wenn man "</"tippt das zuletzt geöffnete Tag schließt. Weiterhin kann die XML-Insert Funktion nützlich sein, die einem bei bekannten Formaten ein Dialogfeld zeigt mit Eingabefeldern für alle möglichen Attribute eines Elements.

Beanshell Scripting JEdit kann durch Java Beanshell Skripte vollständig gesteuert und erweitert werden.

Nur in einem Punkt kann JEdit seine (Java-)Herkunft nicht verleugnen: Bei der Performance. Ein 500 MHz getakteter Prozessor mit 128 MB RAM sind für ein angenehmes Arbeitsgefühl die absolute Untergrenze. Auf langsameren Systemen gestaltet sich die Arbeit mit JEdit deshalb recht zäh.

5.3.2 Ant

Ant ist ein neues Build-Werkzeug für die Java-Anwendungsentwicklung, das plattformübergreifend konzipiert ist und als Format für Build-Dateien XML verwendet. Für Java-Programmierer ist Ant eine gute Alternative zu "make", da das Tool speziell auf die Erfordernisse des Build-Prozesses in der Java-Umgebung zugeschnitten ist. Ant wurde im Rahmen des Jakarta-Projekts der Apache Software Foundation entwickelt und ist Open Source.
Weiterführende Literatur zu Ant findet sich in Stefan Edlichs Buch "Ant - kurz & gut"([19]).

5.3.3 XML

XML, die Extensible Markup Language, ist eine Schlüsseltechnologie für zukünftige Web-Anwendungen. Dieser mächtige W3C (World Wide Web Consortium)-Standard erlaubt es, Daten strukturiert zu speichern und plattformunabhängig auszutauschen.

XML ist äußerst flexibel, da jeder seinen unterschiedlichen Bedürfnissen entsprechend, innerhalb bestimmter Regeln, eigene Datenstrukturen kreieren kann. Diese für Menschen leicht lesbaren Daten können dann zwischen den verschiedensten Anwendungen, Betriebssystemen und Geräten portiert werden. Beispielsweise kann mit Hilfe von XSL-Stylesheets XML in HTML oder andere Präsentationssprachen umgewandelt werden.
Weiterführende Literatur zu Ant findet sich in Robert Ecksteins Buch "XML - kurz & gut, 2. Auflage"([20]).

5.4 Orientierung

Nachdem eine Einschätzung des Testframeworks vorgenommen wurde, soll eine Abgrenzung hierzu durchgeführt werden.

5.4.1 Projekteinschätzung

Um die Elemente der Qualitätssicherung und des Softwaretestens skalieren zu können, ist eine Einschätzung des Projekts erforderlich. Dies soll nach den Ressourcen Budget, Zeit sowie nach Komplexität der Software geschehen.
Das JLiPSD Projekt hat ein eng begrenztes Budget. Die Entwicklungszeit soll auf die Länge einer Diplomarbeit begrenzt, eine eventuelle Weiterentwicklung jedoch unbegrenzt sein.
Das JLiPSD Projekt hat circa 14.000 Zeilen Java Quellcode. Es wird daher als ein Projekt von mittlerer Größe betrachtet.

5.4.2 Abgrenzung

Aufgrund des engen Budgets werden keine kommerziellen Werkzeuge verwendet bzw. untersucht. Als beispielhafter Vertreter der kommerziellen Testwerkzeuge sei hier JProbe-Coverage ([21]) genannt. JProbe Coverage identifizert ungetesteten Programmcode und dessen Ausmaß, garantiert umfassende fehlerfreie Testläufe, verwirklicht Gesamtüberdeckungsergebnisse auch bei mehrfachen Testläufen und bietet druckbare Ergebnisse in HTML oder Textfassung (weitere kommerzielle Vertreter: JTest, JVerify, JCover).

5.5 Untersuchung zur Verfügung stehender Werkzeuge

Das Testframework soll neben einem Klassentestwerkzeug auch ein Überdeckungswerkzeug enthalten. In diesem Abschnitt wird ein Überblick, über die frei verfügbaren Klassentestwerkzeuge sowie die Überdeckungswerkzeuge (coverage tools) des World Wide Web gegeben. Die ausgewählten Werkzeuge werden im nächsten Kapitel dokumentiert.

5.5.1 JUnit

JUnit ([23]) ist ein kleines, mächtiges Java-Rahmenwerkzeug zum Schreiben und Ausführen automatischer Unit Tests. Die Software ist frei und im Kern von Kent Beck und Erich Gamma geschrieben. Die Tests werden direkt in Java kodiert, sind selbstüberprüfend und damit wiederholbar. Testfälle können mit JUnit sehr einfach organisiert und über eine Bedienungsoberfläche ausgeführt werden. Als professioneller Softwareentwickler erhält man mit JUnit ein gebrauchsfertiges Rahmenwerkzeug, um Testprozesse konsistent und konsequent zu automatisieren. Da man das Rahmenwerkzeug mitsamt seinem Quellcode erhält, kann man es auch einfach selbst erweitern. Historisch ist JUnit ein Nachkomme eines ähnlichen Rahmenwerkzeugs für Smalltalk ([22]). Desweiteren ist JUnit Open Source Software unter IBM Common Public License und mittlerweile zum Quasistandard als Java-Unittesting Werkzeug geworden. Dies zeigt sich sowohl in vielen einführenden und weiterführenden Artikeln, als auch in den für JUnit erhältlichen Erweiterungen und der beginnenden Akzeptanz in zahlreichen Open-Source Projekten (Beispielsweise in Apaches TomCat).

5.5.2 NoUnit

NoUnit ist ein kleines, frei erhältliches Überdeckungswerkzeug für JUnit Unittests und wird von Paul Browne ([25]) entwickelt. Es generiert einen HTML Report aus dem Programmcode. Hierbei erzeugt NoUnit ein XML Dokument aus dem Java Bytecode. Dieses XML Dokument wird daran anschließend nach HTML transformiert und liegt als Ergebnisreport zur Auswertung bereit. Der NoUnit Ergebnisreport zeigt dem Tester, welche Methoden des Projekts getestet, teilweise getestet oder ungetestet sind. Dieser Report liefert also keine Ergebnisse bezüglich der Anweisungs-, Zweig-/Entscheidungs- oder Bedingungsüberdeckung (siehe Abs. 3.8.2). Außerdem ist es für den Tester nicht immer nachvollziehbar, warum eine bestimmte Methode

im Programmcode als nicht vollständig getestet angezeigt wird. Ein Grund hierfür ist, daß NoUnit momentan nur als Beta Version zur Verfügung steht, welche zudem recht unzuverlässig ist. Beispielsweise liefert der NoUnit Report als Ergebnis für den Beispieltest der JUnit Homepage[1] ([23]), daß keine Methode hieraus auch nur teilweise getestet worden ist. Daran ist erkennbar, in welcher frühen Entwicklungsphase NoUnit, mit all seinen Kinderkrankheiten bzw. Fehlern (Bugs), noch steckt.

5.5.3 Quilt

Quilt ist ein weiteres Überdeckungswerkzeug, welches wie NoUnit auf Java Bytecode operiert. Die Software ist frei erhältlich und wurde von David Dixon-Peugh und Tom Copeland entwickelt. Quilt stellt Überdeckungsinformationen von JUnit Unittests zur Verfügung. Momentan beschränken sich diese Überdeckungsinformationen auf die Anweisungs- und Zweig-/Entscheidungsüberdeckung. Als Ergebnis eines Testlaufs mit Quilt erhält der Tester ein HTML Ergebnisreport. Ein positiver Aspekt des Quilt Ergebnisreports ist, daß Exceptions, welche während des Testlaufs aufgetreten sind, ans Ende des Reports mit stackTrace und Referenz zum fehlgeschlagenen Test angehängt werden. Dies ist besonders bei einer Fehlersuche sehr hilfreich. Quilt befindet sich momentan auch noch in der Entwicklungsphase, wie die aktuelle Version Alpha 0.4 verrät und die Homepage ist unter Sourceforge zu finden ([24]). Seltsamerweise findet man auf diesen Seiten keine Downloadmöglichkeit von Quilt. Weiteres intensives suchen im Internet brachte auch keine Downloadmöglichkeit zum Vorschein, sodaß dieses grundsätzlich ansprechende Werkzeug auch nicht verwendet werden konnte.

5.5.4 Clover

Clover ist ein Überdeckungswerkzeug für Java Programme und entdeckt somit Teile des Programmcodes, welche nicht ausgeführt werden. Diese Funktionalität wird benutzt um festzustellen, wo Programmcode nicht adäquat durch Tests abgedeckt wird. Als Ergebnis stellt Clover vielfältige Reportmöglichkeiten zur Verfügung, welche sehr einfach und übersichtlich erfaßt werden können. Die Hauptfunktionen von Clover sind:

- Enge Verflechtung mit dem Jakarta Ant Build-Werkzeug. Wenn Ant verwendet wird, um z.B. das anstehende Projekt zu erstellen, kann Clover schnell und einfach integriert und Clover Operationen Teil des Entwicklungsprozeßes werden.

[1] Der berühmte Money Beispieltest der JUnit Homepage

- Präzise, konfigurierbare Überdeckungsaufzeichnung. Clover erfaßt Ausführungs-, Zweig-/Entscheidungs- sowie Methodenüberdeckung.
- Ergebnisreport wahlweise als Text, XML/HTML oder Swing Oberfläche auswählbar. Zusätzlich wahlweise Exklusion der einzelnen Überdeckungsarten im Ergebnisreport möglich.

Clover befindet sich momentan in der Version Beta 0.6 und ist zumindest bsi zum jetzigen Stand frei erhältlich. Laut der Clover Homepage ([26]) ist es zukünftig möglich, daß neue Versionen nicht mehr kostenfrei zu erhalten sind.

5.5.5 Gretel

Gretel ist ein frei verfügbares Überdeckungstestwerkzeug für Java Programme. Entwickelt wurde Gretel von Carl Howells. Die aktuelle Version 1.0rc2 beinhaltet die Erzeugung von Anweisungsüberdeckungsdaten.
Der Hauptunterschied zwischen Gretel und anderen Überdeckungsmonitoren liegt darin, daß Gretel residuale Überdeckungstests überwacht. Das heißt, nachdem ein Programmlauf, welcher von Gretel instrumentiert wurde beendet ist, wird Gretel das Programm reinstrumentieren und die Instrumentierung der Teile löschen, welche häufig ausgeführt wurden. Die meisten Programme laufen die längste Zeit in ein paar abgegrenzten Bereichen, welche einfach überdeckt werden können. Residuale Reinstrumentierung durch Gretel reduziert hierbei den Geschwindigkeitsnachteil der normalen Überdeckungstestwerkzeuge, welche immer wieder neu gestartet werden müssen, um selten ausgeführte Zeilen im Programmcode zu testen bzw. zu überdecken. Weitere Informationen zu Gretel finden sich unter ([27]).

5.5.6 Jester

Jester, geschrieben von Ivan Moore ([28]) ist ein Überdeckungstestwerkzeug für JUnit Unittests. Dieses frei erhältliche Produkt findet Programmzeilen im Code, welche von Tests nicht überdeckt bzw. ausgeführt werden. Wenn Jester gestartet wird, ändert es den Programmcode und startet danach die Tests. Falls die Tests erfolgreich sind, liefert Jester einen HTML Report bezüglich der geänderten Stellen im Programmcode. Jester wird auch Mutations Testwerkzeug (Mutation Testing Tool) genannt, da es den Programmcode ändert.
Jester ist unterschiedlich im Vergleich zu den konventionellen Überdeckungswerkzeugen, da es Programmcode findet, welcher beim Testlauf ausgeführt, aber noch

nicht getestet wurde.
Dieses Werkzeug ist jedoch nicht als Ersatz für konventionelle Überdeckungswerkzeuge, sondern eher als alternative Herangehensweise an die Überdeckungsproblematik gedacht.

Kapitel 6

Design des Testframeworks

In diesem Kapitel werden die Anforderungen an das Design des Testframeworks erarbeitet. Hierbei wird der Entwurf eines Testprozesses vorgestellt. Die anschließende Auswahl einer geeigneten Metrik sichert die Meßbarkeit der Testgüte. Daraufhin werden die Elemente der Testbeschreibung spezifiziert. Abschließend beschäftigt sich das Kapitel mit der Vorstellung einiger Testmuster (Testpatterns) von Binder für Klassen- und Integrationstests.

6.1 Entwurf eines Testprozesses

Nachfolgend wird der Testprozeß entworfen, der sich an konventionellen Methoden ausrichtet und mit den Software Entwicklungszyklen verträglich ist.

6.1.1 Vorgehensweise

Auch wenn recht wenig Dokumentation vorliegt, kann man eine vorliegende Implementierung testen. Man benutzt in diesem Fall die Dokumentation des Programms um Testanforderung und Testspezifikation abzuleiten. Unter Dokumentation soll alles verstanden werden was zur Verfügung steht, vom Benutzerhandbuch über Kommentare im Programmcode bis zu Einzelgesprächen mit dem Programmierer des Projektes ([29]). Mit Hilfe der Dokumentation und Spezifikation kann der Tester nun die funktionalen und strukturellen Tests erstellen. Auch wenn das Programm bereits vorliegt, sollte man auf funktionale Tests ausreichend Gewicht legen.
Wenn man nur darum bemüht ist, strukturelle Tests mit hoher Überdeckungsrate zu erhalten ohne sich gegen Programmspezifikationen abzusichern, erfüllt man Verifikation ohne Validation. Dies bedeutet, daß das Programm zwar korrekt arbeitet, jedoch nicht das korrekte Programm ist, welches gewünscht wird ([30]).

6.1.2 Erstellen der Testdokumente

Im JLiPSD Projekt steht nur die Dokumentation des Programmierers in gedruckter Form zur Verfügung. Die in C-Quellcode integrierte LiPSD Dokumentation kann, wenn überhaupt, nur mit hohem Zeitaufwand verwendet werden. Das Erstellen von Testdokumenten sollte trotzdem einigermaßen leichtfallen, da Einzelgespräche mit dem Programmierer jederzeit durchführbar sind.

6.1.3 Ablauf des Testprozesses

Es muß sowohl dem nachträglichen Testen, genauso wie dem Testen eventuell zukünftiger Erweiterungen Rechnung getragen werden. Für die Weiterentwicklung des JLiPSD können vielfältige Modelle zum Einsatz kommen.
Der Testprozeß gliedert sich für eine vorliegende Implementierung in Testplanung, Stylephase, Testphase und Testauswertung auf. Die Testplanung und die Stylephase, in der die Implementierung testbar gemacht wird, sind Eingangsvorraussetzungen für die Testphase. Die Testphase wird durch Ausführung eines Überdeckungswerkzeugs, in Kombination mit einem Klassentestwerkzeug gestartet und mit der Testauswertung beendet. Die Testphase kann hierbei ein oder mehrmals durchlaufen werden.

Testplanung

In der Testplanung wird die Implementierung nach bestimmten Kriterien (z.B. nach Packages) in Testsuiten und Untersuiten eingeordnet und diesbezüglich werden die Testsuiten erstellt bzw. später ergänzt.

Stylingphase

Im Hinblick auf die Testbarkeit der Implementierung und der allgemeinen Qualitätsverbesserung der Software, wurde eine Stylingphase eingeführt. In der Stylingphase wird die Implementierung auf ihre Testbarkeit untersucht und gegebenenfalls durch Rücksprache mit dem Programmierer verbessert. In dieser Phase werden zusätzlich Supportverzeichnisse für die Tests angelegt. Desweiteren werden konzeptionelle Testanpassungen am Programmcode vorgenommen, da zukünftige Funktionalitätsänderungen des Programms nur durch entsprechende Änderungen am Programmcode erkennbar sind. Hierzu ein Beispiel: Eine leere Methode kann, wenn sie irgendwann implementiert wird, die Funktionalität einer Klasse drastisch ändern. Um diese Änderung überhaupt zu entdecken, muß eine solche Methode zwingend eine Exception werfen, welche im Test abgeprüft werden kann.

Testphase

Während der Testphase wird besonderes Augenmerk auf die bedienungsfreundliche Oberfläche gelegt. Hierzu zählt zum Beispiel ein Fortschrittsbalken, welcher anzeigt wie weit der Testlauf fortgeschritten ist. Zusätzlich wird verlangt, daß sowohl einzeln ausgewählte Tests, als auch alle Tests zusammen ausgeführt werden können.

Testauswertung

In der Testauswertung werden unter Analyse der Testergebnisse Aussagen über die Implementierung auf einer graphischen Benutzeroberfläche oder in einem Dokument festgehalten. Nach der Analyse können dann, falls notwendig, entsprechende Änderungen am Programmcode durchgeführt und die Testphase erneut gestartet werden.

6.2 Metriken

In "Softwaremetriken"(siehe Abs. 2.6) wurden bereits verschiedenen Ansätze vorgestellt. In diesem Kapitel werden nun Metriken für den produktbezogenen Ansatz ausgewählt und im Testframework des JLiPSD ihre Verwendung finden. Zur Einschätzung der Testgüte soll Überdeckung genutzt werden. Zur Komplexitätsmessung des Programmcodes soll die Metrik "lines of code"in brauchbarer Weise zur Verfügung gestellt werden.

Lines of Code

Lines of Code ist zwar eine einfache und relativ ungenaue Metrik ([30]), aber schnell verfügbar und weit verbreitet ([13]). Sie kann, wenn sie richtig berechnet wird, zu Vergleichen mit anderen Projekten herangezogen werden. Diese Metrik ist in den zur Verfügung stehenden Überdeckungswerkzeugen bereits integriert.

6.3 Elemente der Testbeschreibung

Viele Tests, bis auf die einfacheren Unit-/Klassentests verwenden Vertreter-Objekte (Mock-Objekte). Diese Objekte simulieren das Verhalten der Objekte, die sie vertreten sollen. Jeder Test kann in eine Reihe von Testfällen (Unittests) aufgegliedert werden, d.h. jedes getestete Objekt beinhaltet einen Test mit mehreren Testfällen.

Innerhalb dieses Tests werden Methoden benötigt, welche jeden einzelnen Testfall in einen initialen Zustand versetzen. Genauso muß nach jedem Testfall die Möglichkeit bestehen, bestimmte Systemzustände zu beenden bzw. rückgängig zu machen, beispielsweise durch schließen einer Internetverbindung. Nachfolgend wird auf die Inhalte eines Testfalls, die Möglichkeit einer Vertreterfunktionalität, die Platzierung einer Testfallbeschreibung sowie auf absehbare Einschränkungen beim Testen eingegangen.

6.3.1 Inhalte eines Testfalls

Elemente eines Testfalls sind Testbezeichner, Testspezifikation, Testfallverifikation, globale, private und Package private Variablen.
Der Testbezeichner ist für alle Tests einheitlich. Zusammen mit dem eigentlichen Namen für den Test bildet er ein Wort, welches den Test eindeutig beschreibt. Ein Testbezeichner beginnt mit dem Attribut “test“und dem darauf folgenden Testnamen, z.B. “ToString()“. Er lautet in diesem Beispiel also “testToString()“.
Die Testspezifikation beschreibt verbal, wie ein Test ablaufen soll, erwartete Ergebnisse, Komplikationen und ähnliche aufschlußreiche Hinweise. Wenn der Testablauf aus der Testbeschreibung oder dem Testbezeichner offensichtlich ist, z.B. “testToString()“, kann die Spezifikation weggelassen werden.
Die Testfallverifikation soll sicherstellen, ob ein Test bestanden hat oder nicht. Dabei soll in einem Testfall die Möglichkeit bestehen mehrfach Verifikationen durchzuführen.

6.3.2 Vertreterfunktionalität

Im Falle der objektorientierten Programmierung ist es notwenig Vertreter, sogenannte Mocks bzw. Mock-Objekte, für komplexe Objekte zu erzeugen. Diese Vertreterfunktionalität wird als Bedingung vorausgesetzt, da ohne sie sinnvolles Testen eines komplexen Projektes nahezu unmöglich ist. Wichtig bei diesen Mocks ist, daß man sie beim Erstellen eindeutig kennzeichnet, damit im späteren Verlauf des Projektes keine Verwechslungen mit den eigentlichen Objekten auftreten. Die Benennung einer Mock Klasse wird dadurch gekennzeichnet, daß der Klassenname mit dem Wort “Mock“beginnt (z.B. MockTupleSpace).

6.3.3 Plazierung einer Testfallbeschreibung

Die Testfallbeschreibung soll vor der eigentlichen Testmethode in Java-Kommentare eingebettet werden. Sie soll so kurz wie möglich gehalten werden, damit die Tests

nicht unüberschaubar werden. Bei offensichtlichen Testfällen, welche beispielsweise durch ihre Testfallnamensgebung eindeutig beschreiben was sie prüfen, kann auf eine Beschreibung verzichtet werden.

6.3.4 Absehbare Einschränkungen

Es sind bereits einige Einschränkungen in der Formulierungsfreiheit beim Entwickeln der Tests abzusehen.
Ein wichtiger Aspekt ist, daß sicherlich nicht jedes Objekt gemockt werden kann, d.h. es kann nicht immer ein Vertreter geschrieben werden. Bei manchen Objekten ist es einfacher einen Integrationstest zu fahren, als ein komplexes Objekt so zu "mocken", daß es sich auch sinnvoll verhält.
Eine weitere Einschränkung in der Objektorientierung ist, daß manche Methoden durch Zugriffsbeschränkungen entweder gar nicht oder nur indirekt getestet werden können. Desweiteren gibt es auch Methoden, welche keine Rückgabewerte liefern und zusätzlich intern keine globalen Objekte verändern.
Auch abstrakte Klassen stellen in gewisser Weise eine Einschränkung dar. Diese können nur in abgeleiteter Form getestet werden. Die gleiche Problematik ergibt sich auch bei Schnittstellen (siehe Abs. 4.1.2).

6.4 Testmuster für Klassentests nach Binder

Methoden können Einschränkungen unterliegen, in welchem Objektzustand oder in welcher Reihenfolge sie Verwendung finden. Danach richtet sich auch, wie sie getestet werden. Robert V. Binder unterscheidet anhand dieser Kriterien fünf Arten von Klassen: Nonmodale, Unimodale, Quasimodale, Modale und Unterklassen. Jede Art hat ihre eigenen Fehlerquellen und dementsprechend müssen sie getestet werden. Hier sollen erste grobe Richtlinien dafür vorgestellt werden. Binder beschreibt noch detailliertere Muster (Patterns) für die Entwicklung der Tests als hier aufgeführt ([31]). Sinn dieser Klassifizierung ist es, bekannte Erfahrungen wiederverwenden zu können. Die Muster (Patterns) liefern ein Schema, mit dem sich die Klassen eines Systems kategorisieren lassen und sich je nach Kategorie ein Testverfahren ableiten läßt. Genauso wie Desginmuster (Designpatterns) bei der Erstellung des Systems helfen, erleichtern diese Testmuster die Prüfung desselben.

6.4.1 Nonmodale Klassen

Nonmodale Klassen unterliegen keinen Einschränkungen bezüglich des Objektzustandes oder der Aufrufreihenfolge. Einfache Klassen, die nur Werte abspeichern

und lediglich primitive get- und set- Methoden besitzen sind beispielsweise nonmodal. Beim Testen können alle Methoden einzeln getestet werden.

6.4.2 Unimodale Klassen

Dies sind Klassen, deren Methoden nur in festgelegten Reihenfolgen aufgerufen werden können. Ein Beispiel wäre eine Klasse, die ein Kartenspiel simuliert mit einer Methode für jeden Spieler. Die Spieler kommen immer in einer festgelegten Reihenfolge zum Zug, d.h. die Methoden dürfen nur in dieser Reihenfolge aufgerufen werden.
Für den Test bedeutet das, daß alle Methoden in jeder Aufrufsequenz getestet werden müssen.

6.4.3 Quasimodale Klassen

Quasimodal sind Klassen, deren Methoden nur bei bestimmten Objektzuständen aufgerufen werden können. Ein Beispiel dafür ist eine Klasse, die eine Warteschlange (queue) realisiert, die beide Methoden add und remove besitzt und in den beiden Zuständen leer (empty) und nicht leer (not empty) sein kann. Die Aufrufreihenfolge unterliegt keinen Einschränkungen, jedoch kann remove nur ausgeführt werden, wenn die Warteschlange sich nicht im Zustand leer befindet.
Quasimodale Klassen kann man durch einen Zustandsübergangsgraphen beschreiben. Beim Test sollten sämtliche Zustandsübergänge getestet werden. Abbildung 6.1 zeigt einen solchen Graphen für obiges Beispiel.

6.4.4 Modale Klassen

Bei diesen Klassen muß der Zustand der Objekte und die Reihenfolge von Methodenaufrufen berücksichtigt werden. Ein Beispiel dafür ist eine Klasse zur Simulation einer Gangschaltung eines Motorrads mit Rückwärtsgang: Die Reihenfolge der verschiedenen Gänge ist vorgeschrieben, genauso wie der Zustand der durch die Geschwindigkeit bestimmt wird.
Der Test läßt sich wieder nur aus der Abdeckung aller Zustandsübergänge im zugehörigen Zustandsübergangsgraphen ableiten. Der exakte Unterschied zwischen quasimodalen und modalen Klassen ist unter Binders detaillierter Patternbeschreibung ([31]) nachlesbar.

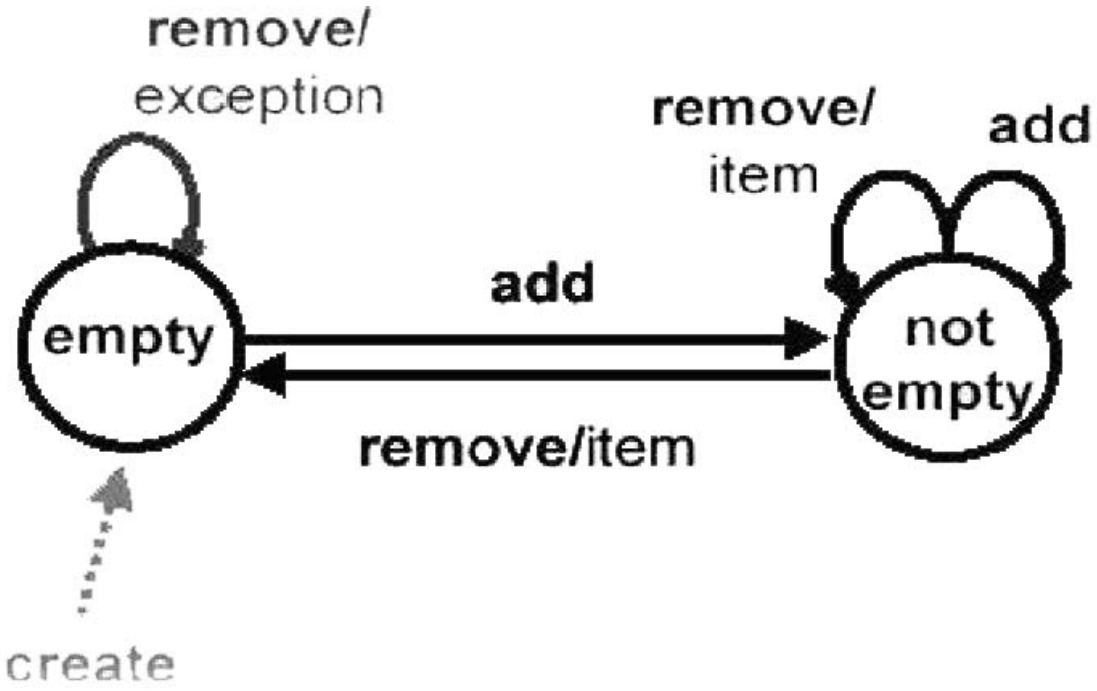

Abbildung 6.1: Zustandsübergangsgraph einer Warteschlangen Klasse

6.4.5 Unterklassen

Daß neue und auch redefinierte Methoden in abgeleiteten, sogenannten Unterklassen getestet werden müssen, erscheint intuitiv einsichtig. Doch auch unveränderte Methoden einer bereits fertig getesteten Basisklasse müssen in Unterklassen neu getestet werden. Andere überschriebene Methoden können von der unveränderten Methode benutzte Objektattribute verändern oder die unveränderte Methode ruft direkt oder indirekt überschriebene Methoden auf, wodurch sich der Kontext in dem die Methoden ablaufen verändert. Dadurch können neue Fehler entstehen oder bereits bestehende Fehler zum Vorschein kommen.
Daher beginnt man den Test bei der Wurzel der Klassenhierarchie und testet dann nacheinander die Unterklassen in der jeweils eins tieferen Hierarchieebene. Hierbei werden bei geerbten und unveränderten Methoden der Basisklasse die dazugehörigen Testfälle erneut ausgeführt. Für neue und redefinierte Methoden müssen neue Testfälle erstellt und ausgeführt werden.

6.5 Testmuster für Integrationstests nach Binder

Es gibt sehr zahlreiche Ansätze zur Integration beim objektorientierten Testen. Robert V. Binder beschreibt neun verschiedene Muster (Patterns) für den Integrationstest. Hier soll lediglich ein grober Überblick über die einzelnen Muster gegeben werden:

Big Bang Integration Alle Komponenten zusammen werden auf einmal getestet.

Bottom-up Integration Integration der Komponenten von den Blättern des Abhängigkeitsbaums zur Wurzel.

Top-down Integration Integration der Komponenten von der Wurzel des Abhängigkeitsbaums zu den Blättern.

Collaboration Integration Integrationsreihenfolge entsprechend des Zusammenwirkens und der Abhängigkeiten.

Backbone Integration Kombination von Top-down Integration, Bottom-up Integration und Big Bang Integration.

Layer Integration Einteilung des Systems in verschiedene Schichten, die entweder Top-down oder Bottom-up integriert werden.

Client/Server Integration Einteilung des Systems in Client- und Serverkomponenten die schrittweise integriert werden.

Distributed Integration Integration verteilter Komponenten (Netzwerkanwendungen).

High-frequency Integration Häufige Wiederholung der Integration.

Detailliertere Informationen zu den Mustern für Integrationstests finden sich unter Robert V. Binder ([31]).

Kapitel 7

Entwicklung

Dieses Kapitel beschreibt hauptsächlich den Aufbau des Testframeworks. Hierbei werden die angesprochenen Aspekte des vorangegangenen Kapitels umgesetzt.

Der Hauptteil des Kapitels stellt das Testframework, die darin verwendeten Werkzeuge und die grundlegenden Mechanismen eines Testablaufs vor. Hierbei sei nochmals darauf verwiesen, daß nur kostenfreie Werkzeuge (Freeware) verwendet werden. Abschließend werden verschiedene Techniken zur Entwicklung *guter bzw. effizienter* Testfälle vorgestellt.

7.1 Werkzeuge des Testframeworks

Das Testframework für JLiPSD beinhaltet, wie schon in Abschnitt 5.4 beschrieben ein Klassentestwerkzeug sowie ein Überdeckungswerkzeug. Diese beiden Werkzeuge werden anschließend ausgewählt.

7.1.1 Auswahl eines Klassentestwerkzeugs

Die Suche nach einem ansprechenden Klassentestwerkzeug verlief sehr einseitig. Das bekannteste Klassentestwerkzeug für Java, namens JUnit ([23]), wurde ausgewählt. Dies liegt einerseits daran, daß es ein sehr ausgereiftes Produkt ist und andererseits daß es ansonsten momentan kein vergleichbares Konkurrenzprodukt gibt, welches auch kostenlos angeboten wird.
Um einen Einblick in die Welt des Unittestens mit JUnit Version 3.8.1 zu erlangen, folgt eine kurze Einführung in dieses Klassentestwerkzeug.

Kurze Einführung in JUnit

Das JUnit-Klassentestwerkzeug wurde, wie schon in Abschnitt 5.5.1 erwähnt, von Kent Beck und Erich Gamma entwickelt. Es besteht aus einer Reihe von Klassen die es erlauben, Werte und Bedingungen zu testen welche jeweils erfüllt sein müssen, damit ein Test erfolgreich ist. Darin enthalten sind Vorlagen und programmtechnische Hilfsmittel zur Organistaion von Testfällen in Testsequenzen (Testsuites) und Testrunner-Werkzeuge, die Tests ablaufen lassen und protokollieren. Das Ziel: Testprozesse sollen soweit automatisiert werden, daß sie möglichst ohne manuellen Eingriff wiederholt werden können und nach jeweiligem Kompilieren ausgeführt werden. Alle gesammelten Tests sollen in einem Testlauf ausgeführt werden.
Das Klassentestwerkzeug unterstützt den Ansatz, die Entwicklung einer Testumgebung parallel zur Kodierung ablaufen zu lassen. Ein wesentlicher Vorteil dieses Vorgehens: Hat man erst einmal eine Reihe von Testfällen definiert, können diese immer wieder verwendet werden. Gerade wenn im Sinne von "Refactoring"die Architektur einer Software überarbeitet wird, sollen die existierenden Tests immer noch fehlerfrei laufen. Das Klassendiagramm in Abbildung 7.1 zeigt die wichtigsten Klassen des JUnit Frameworks anhand von UML (Unified Modeling Language). Mit

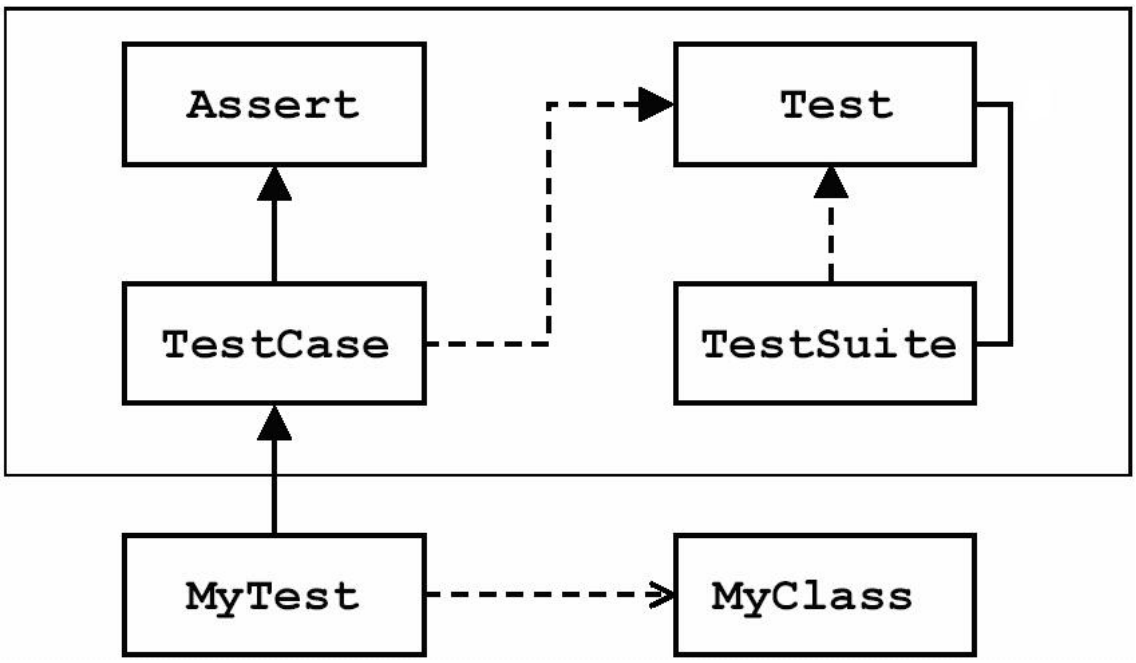

Abbildung 7.1: Klassendiagramm der wichtigsten JUnit Klassen

JUnit können beliebig viele Testfälle in einer sog. "Testsuite"zusammengefaßt und gemeinsam ausgeführt werden. Dazu muß die Methode suite() so implementiert wer-

den, daß die gewünschten Tests auch durchgeführt werden können. Die gewünschten Testfälle werden in der dort festgelegten Reihenfolge ausgeführt. Eine Suite von Tests wird durch ein Testsuite-Objekt definiert, dem beliebig viele Tests und andere Testsuites hinzugefügt werden können. Um eine Testsuite zu definieren, muß eine Instanz der Klasse Testsuite erzeugt und mittels der suite() Methode Testfallklassen hinzugefügt werden. Jede Testsuite enthält eine suite() Methode, in der alle Tests einer Testklasse durch Einbinden dieser Klasse eingefügt werden (siehe Abs. 8.1).
Die Klasse Assert definiert eine Reihe von assert-Methoden, die die vom Benutzer angelegten Testklassen erben und mit denen eine Reihe unterschiedlicher Behauptungen über den zu testenden Code aufgestellt werden können (siehe Abb. 7.1). Die mit assert() kodierten Behauptungen werden von der Klasse Assert automatisch verifiziert. Nicht im Klassendiagramm abgebildet ist die Klasse Testrunner. Sie ruft die Methoden auf. Die Klasse Testcase wird verwendet, um Testfälle um eine gemein-

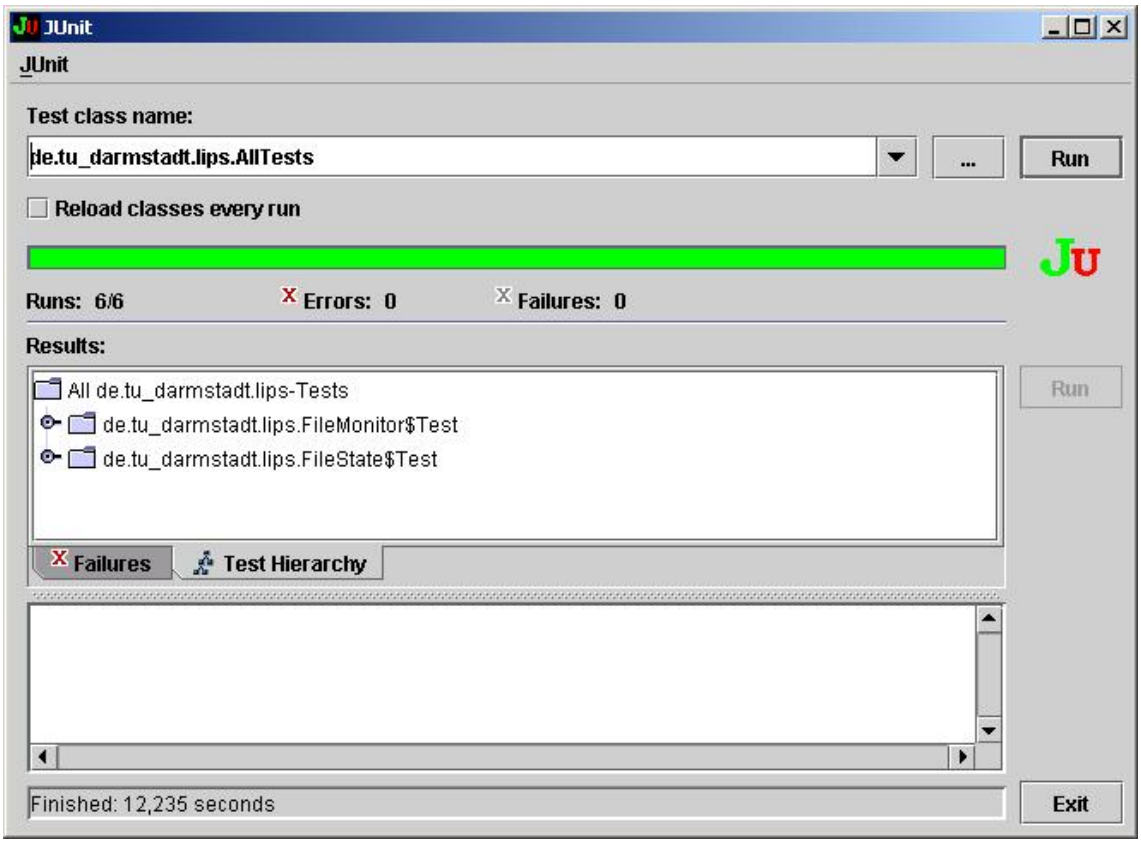

Abbildung 7.2: JUnit Swing-Ergebnisreport

same Menge von Testobjekten zu gruppieren. Jeder Testaufruf zeigt durch einen Rückgabewert oder durch eine Exception den Erfolg oder Mißerfolg eines Tests an.

JUnit bringt dafür eine eigene Oberfläche mit, in der erfolgreiche und fehlgeschlagene Tests visualisiert werden. Dabei hat man die Auswahl zwischen einem rein textuellen oder einem graphischen Ergebnisreport (siehe Abb. 7.2).

7.1.2 Auswahl eines Überdeckungswerkzeugs

Nach der Wahl des Klassentestwerkzeugs wird nun noch ein Überdeckungswerkzeug ausgewählt. Im Gegensatz zu den Klassentestwerkzeugen, gibt es wie in Abschnitt 5.4 aufgeführt mehrere Überdeckungswerkzeuge. Die Wahl fiel hierbei auf Clover ([26]). Clover deckt im Vergleich zu den anderen kostenfreien Überdeckungswerkzeugen am meisten Kontrollfluß ab, nämlich: Methoden-, Anweisungs- sowie Bedingungsüberdeckung. Desweiteren ist Clover ein sehr stabiles Produkt, mit welchem schon große Projekte erfolgreich getestet worden sind. Die verwendete Version 0.6 klingt zwar nicht gerade ausgereift, ist jedoch äußerst zuverlässig.
Um das Arbeiten mit Clover zu erleichtern folgt nun eine kurze Einführung in dieses äußerst interessante Überdeckungswerkzeug.

Kurze Einführung in Clover

Clover ist ein Werkzeug, daß sich in Verbindung mit Apaches Erstellungswerkzeug (build-Tool) Ant einsetzen läßt. Entwickler können mit Clover herausfinden, welche Code-Abschnitte/ Statements abgearbeitet werden. Clover- Reports lassen sich daraufhin in XML, HTML oder über eine Swing-Anwendung betrachten.
Mit Clover kann also jedes Java-Programm getestet werden, nicht nur JUnit als Klassentestwerkzeug, um Überdeckungsergebnisse zu erhalten. Auch bei normalen Applikationen läßt sich feststellen, welche Anweisungen ausgeführt bzw. welche Bedingungen während des Programmlaufs erfüllt werden und welche nicht.
Typischerweise wird Clover per JEdit oder aber per Kommandozeile aufgerufen. Hierfür gibt es vorgefertigte Ant- Ziele (Targets) der build.xml Datei, welche zum Aufruf von Clover verwendet werden. Bei erfolgreichem Kompilen des Programmcodes startet Clover. Hierbei instrumentiert Clover den Programmcode, um Überdeckungsdaten erzeugen zu können. Nach Ende des Cloverlaufs steht der Ergebnisreport, für jede geprüfte Java Klasse separat zur Verfügung, in welcher die nicht durchlaufenen Anweisungszeilen rot eingefärbt sind, um diese besser kenntlich zu machen (Abb. 7.3). Beim validieren mit Clover muß nicht immer erst ein Cloverlauf durchgeführt werden. Clover speichert seine Ergebnisse in einem eigenen Verzeichnis ab und man hat somit die Möglichkeit sich alte Überdeckungsergebnisse zu sichern. Außerdem kann man dadurch ein aktuelles Ergebnis nochmals anschauen ohne erneut z.B. einen kompletten und somit eventuell zeitraubenden Testlauf durchführen

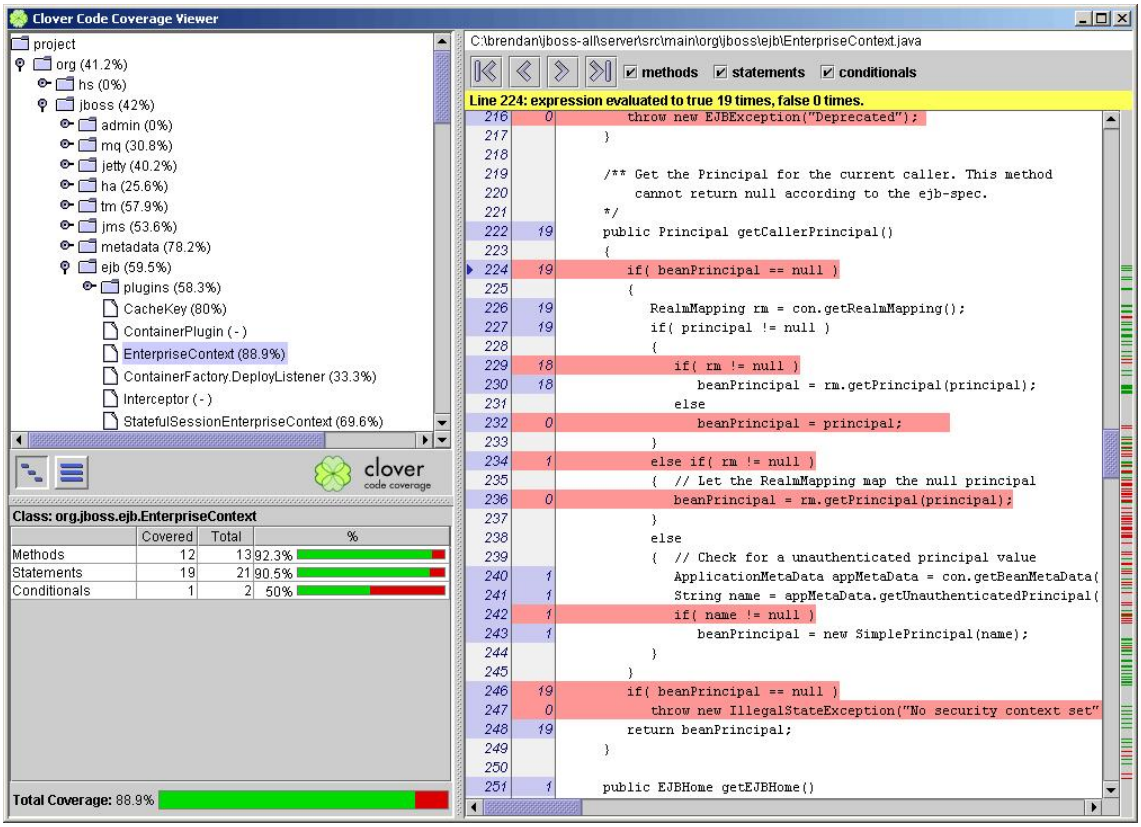

Abbildung 7.3: Clover Swing Ergebnisreport

zu müssen.
Der Clover Ergebnisreport hat weitere sinnvolle Eigenschaften. Beispielsweise kann die Überdeckung für jede einzelne Datei, für Packages oder insgesamt angezeigt werden. Einzelne Überdeckungsarten können während des Ergebnisreports an-/ ausgeschaltet werden. Die Installation von Clover ist auch für nicht geübte Ant Benutzer einfach gestaltet. Eine Installationsanweisung sowie zusätzliche Hinweise zur Verwendung von Clover sind von der Clover Homepage ([26]) zu entnehmen.

7.2 Das Testframework für JLiPSD

In diesem Abschnitt wird das Testframework für JLiPSD vorgestellt. Zuerst wird die Testsuite erstellt. Daran anschließend wird der Bedienungsablauf des Testframeworks vorgestellt.

7.2.1 Erstellen der Testsuites

Um eine Testsuite für JLiPSD zu erstellen, muß zuerst die Hauptsuite erstellt werden. Durch Starten dieser Suite werden später alle Testfälle des JLiPSD ausgeführt. Diese Hauptsuite bindet alle Untersuiten ein. Für jede Package des JLiPSD Projektes wird eine Untersuite erstellt, soweit dies für das Testen des Projektes notwendig ist. In diesen Untersuiten werden die einzelnen Java Dateien, in denen sich Testfälle (Unittests oder Klassentests) befinden, eingebunden. Um das JLiPSD Projekt ausreichend zu testen, müssen nahezu alle Java- Dateien des Projektes getestet und somit Testfälle hierfür erstellt werden.

Erstellen der Hauptsuite

Die Hauptsuite enthält zwei Methoden. Zum ersten die main-Methode, damit sie aufgerufen/gestartet werden kann (siehe Abb. 7.4). Desweiteren wird die Methode suite() benötigt, welche die einzelnen Untersuiten einbindet. Beispielsweise die Untersuite "Com"(Abb. 7.4, Zeile 14). Die Hauptsuite sowie die Untersuiten werden als "AllTests.java"Dateien im jeweiligen Package Unterverzeichnis abgelegt und spiegeln somit namhaft wieder, was in diesen Untersuiten getestet wird, nämlich die Klassen der jeweiligen Packages. Durch addTest() wird eine Untersuite eingebunden (z.B. com-Package Untersuite in Abb. 7.4, Zeile 14) und durch den Aufruf der Methode addTestSuite() wird ein Klassentest zur Suite hinzugefügt (z.B. Klasse FileMonitor in Abb. 7.4, Zeile 8). Die Suite welche in Zeile 12 von Abbildung 7.4 erstellt wird, bekommt einen sinnvollen Namen zugeordnet, damit man sie einfach von den anderen Untersuiten unterscheiden kann. Als Rückgabeparameter liefert die suite() Methode die darin erstellte Haupt oder Untersuite, damit die darin enthaltenen Tests an JUnit übergeben werden können.

Erstellen von Package- Untersuiten

In die jeweiligen Untersuiten des JLiPSD Projekts werden nur Dateien welche Klassentests beinhalten eingebunden. D.h. dort gibt es keine weiteren Untersuiten.
Auch hier wird der Suite, welche in Zeile 10 (Abb. 7.5) erstellt wird, ein sinnvoller Namen zugeordnet, damit man sie einfach von den anderen Untersuiten unterscheiden kann. Auch in dieser Klasse werden die Klassentests zur Suite hinzugefügt, indem man der Methode addTestSuite() die dementsprechende Klasse übergibt (z.B. die Klasse FileReceiver in Abb. 7.5, Zeile 12).

```
01: package de.tu_darmstadt.lips;
02: import junit.swingui.TestRunner;
03: public class AllTests
04: {
05:    public static void main(String args[])
06:    {
07:        junit.swingui.TestRunner.run(AllTests.class);
08:    }
09:
10:    public static junit.framework.Test suite()
11:    {
12:        junit.framework.TestSuite suite =
                new junit.framework.TestSuite("All JLiPSD-Tests");
13:        /* packages */
14:        suite.addTest(de...com.AllTests.suite());
15:        suite.addTest(de...tools.AllTests.suite());
16:        suite.addTest(de...fileTransfer.AllTests.suite());
17:        /* classes */
18:        suite.addTestSuite(de...FileMonitor$Test.class);
19:        suite.addTestSuite(de...FileState$Test.class);
20:
21:        return suite;
22:    }
23: }
```

Abbildung 7.4: Die Test Hauptsuite des JLiPSD

Erstellen der Testfälle

Beim Erstellen der Testfälle für das JLiPSD Projekt stößt man, ohne vorherig gründliche Überlegung, schnell an die Grenzen des Testens. In der Literatur, z.B. “Unit Tests mit Java“ [18] wird der Programmcode nahezu ausnahmslos von den Testfällen getrennt. Dies ist jedoch nur sinnvoll im Bezug auf die Größe des entstehenden Programms. Nachteilig ist hierbei jedoch, daß Tests und Programmcode nicht beieinander liegen, was die Tests unübersichtlich erscheinen läßt.
Ein großes Problem beim Testen allgemein ist, daß man bei getrennten Dateien (Tests zu Programmcode) keine Zugriffsrechte zu privaten Methoden hat. Um dies umgehen zu können, müssen die Tests als innere Klasse der zu testenden eingefügt werden. Somit haben diese Tests volle Zugriffsrechte auf die Methoden und Variablen (globale, private und package-private Variablen) der zu testenden Klasse. Dadurch sind auch die Testfälle und der zu testende Programmcode wieder in einer Datei vereint.
Der Name des Klassentests soll immer “Test“heißen (Zeile 4 in Abb. 7.6) und der Konstruktor ist diesbezüglich auch immer gleich aufgebaut. Deshalb mußbei der Integration von Klassentests in einer Suite immer “Klassenname$Test.java“eingetragen werden. Optional wird in der “setUp“Methode, welche vor jedem Unittest immer

```
01: package de.tu_darmstadt.lips.fileTransfer;
02:
03: import org.apache.commons.logging.Log;
04: import org.apache.commons.logging.LogFactory;
05:
06: public class AllTests
07: {
08:     public static junit.framework.Test suite()
09:     {
10:         junit.framework.TestSuite suite =
            new junit.framework.TestSuite("All fileTransfer-Tests");
11:
12:         suite.addTestSuite(de...FileReceiver$Test.class);
13:         suite.addTestSuite(de...FileSender$Test.class);
14:         suite.addTestSuite(de...FileTransferClient$Test.class);
15:         suite.addTestSuite(de...FileTransferServer$Test.class);
16:         suite.addTestSuite(de...FileTransferRequestListener$Test.class);
17:         suite.addTestSuite(de...FileTransferServerManager$Test.class);
18:         /* Mock Objects */
19:         suite.addTestSuite(de...MockFileTransferCallback$Test.class);
20:
21:         return suite;
22:     }
23: }
```

Abbildung 7.5: Die Untersuite der FileTransfer Package

aufgerufen wird, der Testfall bzw. dessen Objekte initialisiert. Dies spart Platz und hält die Unittests übersichtlich und minimal bezüglich des Platzbedarfs. Optional kann noch die “tearDown“Methode implementiert werden, welche immer nach der letzten Anweisung eines jeden Testfalls ausgeführt wird. Ein Testfall besteht aus dem Methodennamen “testIrgendwas“und muß, wie schon beschrieben, mit dem Wort “test“beginnen.

7.2.2 Konzept des Bedienungsablaufs

Das JLiPSD Testframework muss nach Erstellen der einzelnen Klassen-/ Unittests immer wieder gestartet werden. Der sinnvolle Bedienungsablauf sieht folgendermaßen aus: Zuerst muß ein Klassen-/ Unittest geschrieben werden, welcher auch in der entsprechenden Untersuite eingebunden werden muß. Danach sollte zuerst getestet werden, ob Fehler beim Testlauf mit JUnit auftreten. Falls alle Testfälle erfolgreich im Sinne des Softwareentwicklers (d.h. sie haben bestanden) sind, sollte ein Testlauf mit Clover erfolgen. Hierbei kann durch Konsultieren der Ergebnisse festgestellt werden, welcher Programmcode noch nicht abgedeckt ist. Falls dies der Fall sein sollte, wird ein weiterer Testfall speziell für diese Überdeckungslücke geschrieben. Nun testet man wieder zuerst mit JUnit. Danach, falls alle Tests erfolgreich laufen, wieder

mit Clover, bis die Überdeckungsrate der zu testenden Klasse perfekt (d.h. 100%) oder nicht mehr verbesserbar ist.
Der Grund warum nicht immer mit Clover getestet werden sollte ist, daß Clover mit fortschreitender Projektgröße zunehmend mehr Ausführungszeit benötigt. Besonders deutlich wird dies bei Tests, welche z.B. bestehende Internetverbindungen testen, da ein Verbindungsauf- bzw abbau Zeit kostet und jeder Testfall diesen aufbauen und auch wieder sauber beenden muß.

7.3 Techniken zur Entwicklung guter Testfälle

Um ein Projekt erfolgreich testen zu können benötigt man nicht nur die dementsprechenden Werkzeuge, sondern auch Techniken zum Erstellen von "guten bzw. effizienten"Testfällen. Dies wird einem Tester von Software meist erst dann klar, wenn er merkt wie sich die von ihm entwickelten Testfälle im Laufe der Zeit qualitativ verbessern. Um diesen Entwicklungsschritt zu verkürzen gibt es 16 Richtlinien zur Entwicklung "guter bzw. effizienter"Testfälle, welche nachfolgend auszugsweise erklärt werden. Die meisten dieser Techniken stammen von Andy Schneider, dessen Bericht in JavaWorld ([32]) veröffentlicht wurde.

7.3.1 Verwendung innerer Testklassen

Bei der Verwendung innerer Klassen macht man sich zwei Vorteile zu nutzen. Erstens hat man damit den Programmcode und die hierzu dementsprechenden Testfälle in einer Datei, was beim nachvollziehen der Tests hilfreich ist. Zweitens kann man durch diese Art der Testanordnung auf die privaten Methoden des zu testenden Programmcodes zugreifen. Außerdem hat man Zugriffsmöglichkeiten auf alle privaten, globalen sowie package-privaten Variablen, was sich als positiv herausstellt, wenn man Methoden testen muß, welche nichts zurückliefern.
Der einzigste Nachteil dieser inneren Testklassen besteht darin, daß man die Tests von Hand löschen muß, falls man sie nicht mehr, z.B. in der Endversion, direkt beim Programmcode stehen haben will.

7.3.2 Behandlung von Ausnahmesituationen

Grundsätzlich sollten alle Ausnahmesituationen (Exceptions), welche geworfen werden können auch getestet werden. Hierbei stößt der Tester dann schnell auf gewisse Grenzen. Sicherlich kann jede Exception erzwungen werden, doch zu welchem Preis?

Dies bedeutet, daß der Faktor Zeit schnell den Nutzen überwiegt. Es sei dem Tester der jeweiligen Software überlassen, in wie weit er Exceptions testet. Das in Abbildung 7.7 aufgeführte Beispiel illustriert eine Test- Exceptionbehandlung, welche aufgefangen und getestet werden soll.
Im Try- Block wird die Methode methodThrowsException aufgerufen. Diese soll eine NullPointerException werfen. Falls sie dies macht, wird die Zeile 6 nicht mehr aufgerufen und der "assert"Methodenaufruf in Zeile 10 (Abb. 7.7) wird dazu führen, daß der Test von JUnit als erfolgreich bewertet wird. Falls die Methode in Zeile 5 jedoch keine Exception wirft, wird in Zeile 6 ein Fehler an JUnit propagiert, welcher dann auch angezeigt wird. Wenn die Zeile 6 nicht existiert und die Methode in Zeile 5 keine Exception wirft, dann würde der Softwaretester vielleicht gar nicht die Kenntnis erlangen, daß hier ein unerwartetes Verhalten auftritt. Dies bedeutet, daß ein Test als erfolgreich bewertet wird, wenn keine Assert-Anweisung ausgeführt wurde.
Falls man eine Exception nicht überprüfen will (z.B. weil sie nahezu nie geworfen wird) empfiehlt es sich, auch im Zuge der besseren Übersichtlichkeit der Testfälle, keine Try-Catch Blöcke zu benutzen, sondern den throws Zusatz an die Methodendeklaration zu heften. Bezugnehmend auf das obige Beispiel wird in Abbildung 7.8 davon ausgegangen, daß die Exception nicht geworfen wird. Durch die Verwendung der throws Anweisung (siehe Abb. 7.8) wird der Testfall deutlich übersichtlicher. Übersichtlichkeit ist eine zusätzliche Richtlinie beim Erstellen von "guten"Testfällen.

7.3.3 Testfälle klein halten

Testfälle sollen immer so klein wie möglich gehalten werden. Dies hängt vor allem mit der sinkenden Übersichtlichkeit zusammen, je größer die Tests werden. Desweiteren ist es oftmals so, daß bestimmte Klassen überarbeitet werden müssen und somit kleine Änderungen auftreten, welche Tests fehlschlagen lassen. Falls dies der Fall sein sollte, muß man diese Tests anpassen. Je größer nun die Testfälle gehalten sind, desto unübersichtlicher sind diese und desto mehr Einarbeitungszeit benötigt man. Ein weiterer wichtiger Aspekt ist, daß je kleiner die Testfälle sind, desto weniger fehleranfällig sind die Tests an sich.

7.3.4 Vermeidung von Seiteneffekten

Testfälle müssen immer reihenfolgenunabhängig getestet werden können. Wenn dies nicht der Fall wäre, also Seiteneffekte bestimmter Testfälle auftreten würden, könnten diese Tests andere Tests negativ beeinflußen oder sogar dazu führen, daß andere bzw. nachfolgende Tests fehlschlagen ([32]).

7.3.5 Zeitunabhängigkeit

Tests mit zeitbehafteten Daten sollten möglichst immer vermieden werden, vor allem wenn die zu prüfende Zeit ungültig werden kann. Zeitabhängige Situationen sollten eher manuell oder programmatisch getestet werden. Oftmals ist es auch einfacher die zu testende Klasse mit einem Mechanismus zum ändern der Zeit zu instrumentieren. Daraufhin kann dann der Test zeitunabhängig ausgeführt werden, ohne daß Daten erneuert/geändert werden müssen ([32]).

7.3.6 Architekturunabhängigkeit

Tests sollten architekturunabhängig laufen können. Insbesondere wenn man Software testet, welche plattformübergreifend eingesetzt werden kann (z.B. Java), ist es von besonderem Interesse die Tests ohne vorherige zeitaufwendige Anpassung sofort ausführen zu können ([32]).

7.3.7 Systemtests

Als Softwaretester sollte man immer mindestens einen Testfall erzeugen, welcher das gesamte System testet. Dies wird dann deutlich, wenn an einer Klasse oder Methode etwas geändert wird. Diese Änderung erkennt man sofort an den Auswirkungen der Änderung auf das System, da dies die Testergebnisse des Systemtests zeigen werden ([32]).

7.3.8 Selbstbeschreibende Namensgebung

Testfälle müssen einen selbstbeschreibenden Namen erhalten. In Abbildung 7.9 wird die Methode “getValue“durch den Unittest in Zeile 6-9 getestet. Der Test erhält den Namen “testGetValue“und ist somit offensichtlich selbstbeschreibend, bezüglich seiner Testaufgabe. Bei diesem Unittest kann somit auf eine Testfalldokumentation verzichtet werden, da diese implizit in der Unittest Methodendeklaration steckt.

```
01: // Unit Tests for ByteConverter
02: /** @author Jochen H"ahnle */
03:
04: public static class Test extends JLipsDTestCase
05: {
06:     private Byte[] ByteArray;
07:     private byte[] byteArray;
08:
09:     public Test(String name)
10:     {
11:         super(name);
12:     }
13:
14:     protected void setUp()
15:     {
16:         byteArray = {17,2,0,1,42};
17:         ByteArray = new Byte[5];
18:         ByteArray[0] = new Byte(byteArray[0]);
19:         ByteArray[1] = new Byte(byteArray[1]);
20:         ByteArray[2] = new Byte(byteArray[2]);
21:         ByteArray[3] = new Byte(byteArray[3]);
22:         ByteArray[4] = new Byte(byteArray[4]);
23:     }
24:
25:     public void testToByteArray()
26:     {
27:         for(int i=0;i<ByteArray.length ;i++ )
28:         {
29:             assertEquals(ByteConverter.
                    toByteArray(ByteArray)[i],byteArray[i]);
30:         }
31:     }
32:
33:     public void testToByteObjectArray()
34:     {
35:         for(int i=0;i<ByteArray.length ;i++ )
36:         {
37:             assertEquals(ByteConverter.
                    toByteObjectArray(byteArray)[i],ByteArray[i]);
38:         }
39:     }
40: }
```

Abbildung 7.6: Unittest Beispiel der Klasse ByteConverter

```
01: public void testFoo()
02: {
03:    try
04:    {
05:       methodThrowsException();
06:       fail("Fehler");
07:    }
08:    catch(Exception e)
09:    {
10:       assertTrue(e instanceof NullPointerException);
11:    }
12: }
```

Abbildung 7.7: Exceptionbehandlung, wenn diese abgefangen werden soll

```
01: public void testFoo() throws NullPointerException
02: {
03:    methodThrowsException();
04:    assertTrue(whatsOever);
05: }
```

Abbildung 7.8: Exceptionbehandlung, wenn diese nicht auftritt

```
01: public int getValue()
02: {
03:  return value;
04: }
05: ...
06: public void testGetValue()
07: {
08:  assertTrue(foo.getValue(),17);
09: }
```

Abbildung 7.9: Beispiel zur Testnamensgebung

Kapitel 8

Das Testframework in der Anwendung

In diesem Kapitel wird die Anwendung des Testframeworks erläutert. Hierdurch wird der Leser mit der Bedienung des Testframeworks vertraut gemacht. Die Verwendung der Testbeschreibungssprache von JUnit wird als bekannt vorausgesetzt, so daß nur auf das Einfügen der Testbeschreibung, das Aktualisieren der Testsuite und den Bedienungsablauf eingegangen wird. Das Kapitel schließt mit den Testergebnissen des JLiPSD Projektes.

8.1 Bedienung des Testframeworks

Die Bedienung des JLiPSD Testframeworks beruht hauptsächlich in der Verwendung des JEdit Entwicklungswerkzeugs. Dort werden alle Funktionalitäten des Testframeworks zur Verfügung gestellt. Hierbei ist egal, ob es das Ziel ist einen neuen Klassentest zu erstellen oder ein JUnit- Testlauf ohne bzw. mit Clover Ergebnissen durchzuführen.
Der nachfolgende Abschnitt beschreibt die allgemeine Verfahrensweise bei der Bedienung des Testframeworks. Hierzu gehört eine dementsprechende Beschreibung der Testfälle, welche es leichter gestalten einen Test nachzuvollziehen. Zusätzlich sollten die Testsuites auf einem aktuellen Stand gehalten werden, da ansonsten keine Testläufe mit allen Testfällen des JLiPSD-Projektes ausgeführt werden. Nach der Testausführung werden die Ergebnisse konsultiert, um die Güte des jeweiligen Testlaufs zu verifizieren und eventuelle Änderungen im Testcode oder Programmcode vorzunehmen. Daran anschließend muß der veränderte Code nochmals getestet werden, durch Wiederholung des Testlaufs.

8.1.1 Einfügen eines Testfalls

Ein neuer Testfall kann zu einem bestehenden Klassentest hinzugefügt werden, indem die Testfallmethode und deren Funktionalität eingefügt wird. Die Testfallmethode muß dabei innerhalb der inneren Klasse korrespondierend zum jeweiligen Klassentest eingefügt werden. Der Testfall wird bei einem nachfolgenden Testlauf automatisch integriert und ausgeführt, d.h. er muß nicht per Hand in einer Suite eingetragen werden.

8.1.2 Einfügen einer Testbeschreibung

Eine wichtige Komponente eines jeden Testfalls ist die Dokumentation, die sogenannte Testbeschreibung. Diese kann sicherlich bei manchen Tests entfallen, wie z.B. bei Tests von Get und Set Methoden. Bei anderen Testfällen empfiehlt es sich eine kurzgehaltene Testbeschreibung zu erstellen. Abhängig von einer "guten"Namenswahl eines Testfalls (z.B. testGetMethode()) kann man eventuell auch bei komplexeren Tests auf eine Testbeschreibung verzichten, falls der Name eindeutig den Testfallablauf beschreibt. Falls eine Testbeschreibung erforderlich wird, sollte man diese in

```
01: public int getIntValue()
02: {
03:    return intValue;
04: }
05: ...
06: // Hier wird die Testbeschreibung eingefuegt
07: // Diese Methode prueft den Rueckgabewert von intValue();
08: public void testGetIntValue()
09: {
10:    assertTrue(foo.getIntValue(),17);
11: }
```

Abbildung 8.1: Platzierung der Testbeschreibung

die bekannten Java Kommentarzeilen einbinden und unmittelbar vor dem eigentlichen Testfall positionieren, wie in Abbildung 8.1 gezeigt wird. Die Dokumentation des Unittests "testGetIntValue"steht unmittelbar vor der Methodendeklaration des Unittests. Hierbei sei deutlich gemacht, daß eine Testbeschreibung keinen Roman beinhalten sollte, da sich dies negativ auf die Übersichtlichkeit auswirkt.

8.1.3 Erstellen eines Klassentests

Soll ein Klassentest hinzugefügt werden, muß dieser in der entsprechenden Testsuite der korrespondierenden Package eingefügt werden, wie z.B. der ByteConverter Klassentest in Abbildung 7.6. Hierbei ist wichtig, daß bei einer etwaigen Neuentwicklung eines Klassentests mindestens ein Testfall (dieser kann zu Beginn auch leer sein!) definiert wird, da der JUnit Testlauf ansonsten fehlschlägt. Der Klassentest wird immer als innerne Klasse in die zu Testende eingefügt. Das Hinzufügen eines Klassentests in einer Testsuite wird im nächsten Unterabschnitt erklärt.

8.1.4 Aktualisieren der Testsuites

Die Testsuite muß prinzipiell immer dann aktualisiert werden, wenn ein neuer Klassentest erstellt wurde. Es hat sich jedoch gezeigt, daß sich das Auskommentieren einzelner Klassentests als sinnvoll erweißt, da große Projekte zu langen Testlaufzeiten führen. Dadurch werden nur einige wenige Tests immer durchgeführt und falls diese positiv verlaufen sollten, muß die Testsuite wiederum aktualisiert (d.h. alle Tests werden wieder einkommentiert) werden, um alle Tests des Projektes verifizieren. In

```
01: package de.tu_darmstadt.lips.tools;
02: public class AllTests
03: {
04:    public static junit.framework.Test suite()
05:    {
06:        junit.framework.TestSuite suite = new
07:         junit.framework.TestSuite("All de...tools-Tests");
08:
09:        suite.addTestSuite(de...LoadInfo$Test.class);
10: //       suite.addTestSuite(de...InputStr$Test.class);
11:        suite.addTestSuite(de...OutputStr$Test.class);
12:        ...
13:        return suite;
14:    }
15: }
```

Abbildung 8.2: Beispiel Testsuite für die tools Package

Abbildung 8.2 ist in Zeile 10 sichtbar, daß dieser Klassentest auskommentiert ist. Wenn dieser auskommentierte Test eine Laufzeit von drei Minuten benötigt, spart man sich pro Testlauf diese Zeit. Wichtig ist jedoch, daß man diesen zu einem späteren Zeitpunkt wieder einkommentiert, da ansonsten eventuelle Seiteneffekte zu anderen Tests nicht sichtbar werden. Falls ein neuer Testfall, beispielsweise der Klasse “LoadInfo“einzufügen ist, so muß dieser wie in Zeile 9 (Abb. 8.2) eingefügt werden. Die Punkte innerhalb der addTestSuite Methode dienen hierbei der Übersicht.

Vor dem eigentlichen Klassennamen muß die komplette Packageadresse angegeben werden, welche hier zu lang ausfällt und durch die Punkte ersetzt wurde. Nach dem Klassennamen muß "$test"vor der ".class"Angabe stehen, da der Klassentest als innere Klasse in der Klasse LoadInfo eingefügt ist.

8.1.5 Testausführung

Mittels der in der build.xml Datei vordefinierten Ziele (Targets) kann in JEdit ausgewählt werden, ob man einen Test mit JUnit oder mit Clover und JUnit durchführen will (siehe Abb. 8.3). Diese Entscheidung hängt davon ab, ob man verifizieren will, wie die Unit-/ Klassentests verlaufen oder aber man per Überdeckungsparameter nicht getesteten Programmcode entdecken will. Nochmals sei erwähnt, daß Clover im Allgemeinen mehr Zeit bis zur Ergebnisbildung benötigt, da es zum Erstellen der Überdeckungswerte den Programmcode instrumentieren muß. Das Testframework wird komplett über JEdit gesteuert und neue Testfälle mittels dieses Werkzeugs implementiert und in die Testsuite eingefügt. Die Bedienoberfläche von JEdit verfügt hierbei über die Möglichkeit, daß Ant Werkzeug zu integrieren und dessen Ziele (Targets) anzuzeigen. Hierfür gibt es am linken Rand der JEdit Oberfläche die Möglichkeit, die Ant Ziele zu visualisieren. Durch Auswahl dieser erscheinen die Ziele im linken Fenster. Hierbei wird zum Testen mit Clover und JUnit *run.with.clover* verwendet. Dabei wird mit Reflection Technik getestet. Zum Testen des Projektes durch JUnit ohne Clover, muß das Ziel *test* ausgewählt werden. Hierbei wird das Projekt vor dem eigentlichen Testlauf automatisch kompiliert. Um den Clover Ergebnisreport zu aktivieren muß eines der drei zur Verfügung stehenden Ziele *clover.emacs.report*, *clover.html.report* oder *clover.swing.report* ausgewählt werden. Die Ziele werden in der build.xml Datei von Clover bereits mitgeliefert und stehen nach der Installation von Clover zur Verfügung. Diese Datei muß nun noch in den Installationspfad von JEdit kopiert werden, damit JEdit die Ziele (Targets) hierzu findet. Wie aus Abbildung 8.3 ersichtlich ist sind noch weitere Ziele definiert, welche jedoch nicht zum Testen relevant sind und teilweise beim Entwickeln des Projektes entstanden, d.h. nicht im Lieferumfang von Clover enthalten sind.
Auf eine Einführung in die Funktionsweise von JEdit wird im folgenden verzichtet, da diese Entwicklungsumgebung äußerst umfangreich ist und die Einführung somit den Rahmen dieser Arbeit sprengen würde. Eine detaillierte Einführung findet sich jedoch auf der Homepage von JEdit ([35]).

8.1.6 Konsultieren der Ergebnisdaten

Es stehen diverse Ergebnisreports zur Verfügung, je nachdem ob mit Überdeckung bzw. Clover oder ohne , d.h. mit JUnit isoliert, getestet wurde. Bei JUnit legt sich der Tester beim Aufruf fest, welche Test- Oberfläche angezeigt werden soll. Bei Clover hingegen kann der Tester nach einem Testlauf entscheiden, welchen Ergebnisreport er gerne generiert hätte.

JUnit Reports

Es gibt drei Möglichkeiten unter JUnit einen Testreport angezeigt zu bekommen. Dieser wird wie vorher erwähnt zur Aufrufzeit eines Testlaufs festgelegt. Dieser Testreport wird zur Laufzeit aktualisiert. Die einfachste Reportart von JUnit ist die Textausgabe der Testergebnisse. Zusätzlich zur textuellen Ausgabemöglichkeit bietet JUnit eine graphische Testoberfläche. Je nachdem ob man eine Awt Oberfläche der Swing Alternative (siehe Abb. 8.4) vorzieht, kann man diese Oberfläche beim Teststart festlegen. Es empfiehlt sich jedoch, die Swing Komponente den beiden anderen Reportmöglichkeiten vorzuziehen, da diese sicherlich die ausgereifteste Variante darstellt. Das Testframework besitzt im Entwicklungswerkzeug JEdit momentan nur die Möglichkeit, die Swing Oberfläche zu verwenden. Geringe Anpassungen vermittels XML in der build.xml Datei von JEdit, können auch durch weniger versierte Benutzer durchgeführt werden, damit die anderen Reports zur Verfügug stehen.
Für die Textausgabe als Report unter JUnit dient eine Batch Schnittstelle. Um diese zu benutzen muß “junit.textui.TestRunner“in der main() Methode der Hauptsuite in Zeile 7 (siehe Abb. 7.4), anstelle von “junit.swingui.TestRunner“, angegeben werden. Diese Batch- Schnittstelle liefert das Testergebnis als Textausgabe. Der JUnit Awt- Ergebnisreport hat die gleiche Funktionalität wie der Swing Report, sieht nur durch die Verwendung anderer Java Komoponenten in seinem Erscheinungsbild unterschiedlich aus. Auf die Illustration des Awt Reports, sowie des Textreports wird an dieser Stelle verzichtet.

Clover Swing Report

Die Oberfläche des Clover Swing Ergebnisreports (siehe Abb. 8.5) unterteilt sich grob in zwei Hälften. Die linke Hälfte beinhaltet die Navigation sowie die Überdeckungsergebnisse der einzelnen sowie der gesamten getesteten bzw. auch ungetesteten Klassen. Durch die Darstellung des Projektes als Baum erhält der Softwarete-

ster schnell einen Überblick über die Gesamtergebnisse, da die Knoten des Baumes die jeweiligen Untersuiten präsentieren. Durch Auswahl eines Blattes kann das Ergebnis eines Klassentests hierzu angezeigt werden. Hierzu wird der Programmcode auf der rechten Seite des Ergebnisreports visualisiert und nicht überdeckte Programmzeilen rot eingefärbt. In der linken unteren Ecke des Reports werden sowohl die Ergebnisse der einzelnen Überdeckungsarten als auch das Gesamtüberdeckungsergebnis einer ausgewählten Suite oder Klasse angezeigt.

Clover HTML/XML Report

Der Clover HTML/XML Report (siehe Abb. 8.6) repräsentiert die Ergebnisse auf Basis einer Internetseite. Durch Auswahl einer Klasse im linken unteren Fenster des Ergebnisreports wird diese im rechten Fenster dargestellt und wie beim Clover Swing Report werden die nicht überdeckten Programmzeilen zur besseren Illustration rot eingefärbt. Die Überdeckungsergebnisse stehen hier über der dargestellten Klasse. Im Gegensatz zur Auswahl einer Klasse, werden bei der Auswahl einer Untersuite (siehe Abb. 8.7) oder der Hauptsuite alle Klassen auf der rechten Fensterseite des Reports durch die Überdeckungsergebnisse präsentiert. Hierbei wird deutlich, welche Klassen bereits vollständig getestet wurden und wo eventuelle Nachbesserungen durchgeführt werden müssen.

8.1.7 Wiederholung von Testläufen

Prinzipiell kann man einen Testlauf jederzeit wiederholen. Der Hauptunterschied von JUnit im Vergleich zu Clover Wiederholungstestläufen wird nachfolgend aufgezeigt.

JUnit

JUnit hat die Fähigkeit, einen Testlauf jederzeit zu unterbrechen sowie einen Testlauf jederzeit wieder zu starten[1]. Deshalb kann bei JUnit auch ein Testlauf jederzeit wiederholt werden. Initial testet JUnit immer alle Testfälle, welche in den Testsuiten eingebunden sind. Nach einem kompletten Testlauf können auch einzelne Tests des Testlaufs wiederholt werden. Dies zeigt sich als hilfreich, weil bestimmte Ausnahmesituationen (z.B. Exception wurde geworfen) im Ergebnisreport von JUnit gekürzt angezeigt werden. Die eigentliche Ausgabe wird jedoch immer noch auf der Ausgabekonsole von JEdit sichtbar.

[1] Vorraussetzung eines Starts ist, daß der vorherige Testlauf gestoppt wurde.

Wichtig ist, daß der Haken auf der JUnit (Ergebnis-) Oberfläche, welcher das wieder einladen der Klassen zu jedem Testlauf betrifft immer ausgeblendet ist. Dies wurde zwingend notwendig, als das Chainsaw-Logging Werkzeug ([34]) eingebaut wurde. Es wird vermutet, daß der Fehler wegen der gleichzeitigen Verwendung des Chainsaw Werkzeugs mit Ant auftritt.

Clover

Beim Testen mit Clover hat man die Möglichkeit die Hauptsuite des JLiPSD aufzurufen. Hierdurch wird dann automatisch JUnit aufgerufen und die Tests durch Clover instrumentiert.
Als Alternative hierzu gibt es die Möglichkeit, die Tests komplett via Reflektion (siehe Abs. 4.3) zu testen (Ant Target twc). Hierbei werden alle Testfälle in den Klassen durch Reflektion herausgefiltert und ausgeführt. Reflektion ist allgemein gefaßt die Fähigkeit während der Ausführung eines Programms, wobei Daten das ausgeführte Programm vertreten, den Programmzustand zu manipulieren ([36]).

8.2 Testergebnisse von JLiPSD

In diesem Abschnitt werden die Ergebnisse der Tests des JLiPSD Projektes vorgestellt. Insgesamt wurden 1049 Tesfälle, bei einer Projektgröße von ungefähr 14.000 Lines of Code (siehe 5.2), erstellt. Hierbei sind 51 Tests enthalten, welche als Integrationstests bzw. Subsystemtests bezeichnet werden. Nachfolgend werden die Überdeckungsergebnisse des JLiPSD Testens vorgestellt. Das Kapitel endet mit der Analyse der gefundenen Fehler beim Testen des JLiPSD Projektes.

8.2.1 Überdeckungsergebnisse

Die Überdeckungswerte beim Testen des JLiPSD Projektes mit JUnit und Clover ergeben folgendes Ergebnis.

98,0 % tools - Package

84,3 % fileTransfer - Package

79,9 % com - Package

Dies ergibt eine Gesamtüberdeckung des Projektes, laut Clover von 81,6 % des getesteten Programmcodes (siehe Abb. 8.8). Dieser Wert ist jedoch nicht korrekt. Der Klassenlader (Classloader) von Ant/Java hat einen Fehler welcher dazu führt, daß JUnit eine *NoClassDefFoundError* Exception wirft, sobald eine Klasse mehr als 5 mal zur Laufzeit instanziiert wurde (Ant Fehler 3158). Insgesamt sind 7 Klassen des JLiPSD Projekt diesbezüglich betroffen.
Durch nachträglich manuelles hinzurechnen der Testergebnisse der betroffenen Klassen, ergibt sich eine Gesamtüberdeckung des Projekts von 85,4 %.

8.2.2 Analyse entdeckter Fehler

Während des Testens wurden 82 Fehler in den 148 Klassen des JLiPSD Projektes gefunden, welche daraufhin durch den JLiPSD Entwickler ([33]) beseitigt wurden. Diese Änderungen wurden nachfolgend verifiziert und erneut getestet, bis keine Fehler mehr nachweisbar waren.
Die folgende, kategorisierte Auflistung der gefundenen Fehlerursachen macht deutlich, wo und wie häufig sich bei der Entwicklung von Software Fehler *einschleichen* können.

Kopieren und Einfügen - Fehler

Die Verwendung von "Kopieren und Einfügen"(copy and paste) ist eine der häufigsten Fehlerursachen. Oftmals wird, vor allem bei Verwendung von abstrakten Klassen und Vererbung, ein Teil eines bereits erstellten Programmcodes kopiert und wiederverwendet (z.B. Get und Set Methoden). Hierbei müssen jedoch meist, wenn auch nur geringfügig, Änderungen durchgeführt werden, damit das bereits geschriebenen Codefragment benutzt werden kann. Im JLiPSD Projekt wurden 17 *Kopieren und Einfügen* Fehler entdeckt und anschließend beseitigt. In Anbetracht der Projektgröße ist diese Anzahl statistisch betrachtet durchaus gering.

Dateibearbeitung

Bei der Benutzung von Dateien, welche im Projekt zur Konfiguration desgleichen Verwendung finden, wurden 9 Fehler entdeckt. Diese Fehler resultierten meist in der Tatsache, daß die Datei nach Verwendung nicht wieder frei gegeben wurde (z.B. Datei war noch in Benutzung, obwohl sie es nicht mehr sein sollte).

Verwendung inkonsistenter Datentypen

Bei der Entwicklung des JLiPSD mußten verschiedenste Protokolle zum Versenden von Nachrichten implementiert werden. Um diese Protokolle zu erfüllen, wurde es nötig diverse Datentypen zu verwenden. Dabei wurden 10 Fehler gefunden, welche durch die Verwendung eines inkonsistenten bzw. falsch gewählten Datentypen auftraten.

Vergleiche

Das Testen von Kontroll- bzw. Vergleichssituationen wurde im JLiPSD Projekt intensiv betrieben. Die 9 hierbei gefundenen Fehler hatten ihre Ursache zumeist in der Verwendung einer falsch gewählten Abbruchbedingung.

Adressfehler

Während des Testens wurde ein Fehler entdeckt, welcher darin begründet war, daß eine falsche Internetprotokoll (IP) Adresse ihre Verwendung fand. Interessanterweise wurde dieser Fehler erst entdeckt, als das komplette Projekt zusätzlich auf einem anderen Rechner getestet wurde.

Ausnahmesituationen

Bei der Behandlung von Ausnahmesituationen (Exceptions) wurden in dem JLiPSD Projekt zwei Fehler gefunden.

Verbindungsfehler

Bei der Benutzung von Internetverbindungen wurden drei Fehler gefunden. Hierbei wurden die benutzten Internetverbindungen zwar korrekt geöffnet und verwendet, jedoch nicht geschlossen was zur Ursache hat, daß ein mehrfaches Öffnen/ Bereitstellen dieser Verbindung fehlschlägt.

Leichtgewichtige Prozesse

Die Programmiersprache Java stellt dem Softwareentwickler leichtgewichtige Prozesse zur Verfügung. Diese Prozesse werden als Fäden (threads) bezeichnet. Jeder dieser Fäden muss somit nicht nur gestartet und ausgeführt, sondern auch beendet werden. Da bei der Verwendung mehrerer Fäden zur Laufzeit nie sicher ist, welcher Faden sich in welchem Zustand befindet, ist es schwierig Programmabschnitte hierzu zu entwickeln. Noch komplexer gestaltet sich die Entwicklung von Testfällen hierzu. Insgesamt wurden 9 Verwendungsfehler entdeckt.

Fehler in der Programmlogik

Insgesamt wurden 7 Fehler in der Programmlogik gefunden. Dabei wurden mehrfach Längenberechnungen sowie das Sortieren von Objekten falsch implementiert.

Kontrollausgaben

Als eine besondere Nachlässigkeit in der Programmierung stellte sich das Finden dreier Fehler während des Testens heraus. Diese Fehler traten immer nur dann auf, wenn die Loggingausgaben (Kontrollausgaben des Software Entwicklers) ausgeschaltet wurden. Als Erklärung für dieses Verhalten fand sich in drei Logginganweisungen darin inkludierte Programmlogik.

Redundanzen

Abschließend wurde durch das Testen zwölfmal redundanter Programmcode entlarvt und beseitigt. Häufig wurden Variablen deklariert und initialisiert, jedoch nicht weiter verwendet. In Anbetracht der Tatsache, daß das JLiPSD Projekt in Sachen Durchsatz und Qualität seinem Original, dem LiPSD, in allen Belangen ebenbürtig sein soll, werden auch diese Redundanzen als Fehler gewertet.

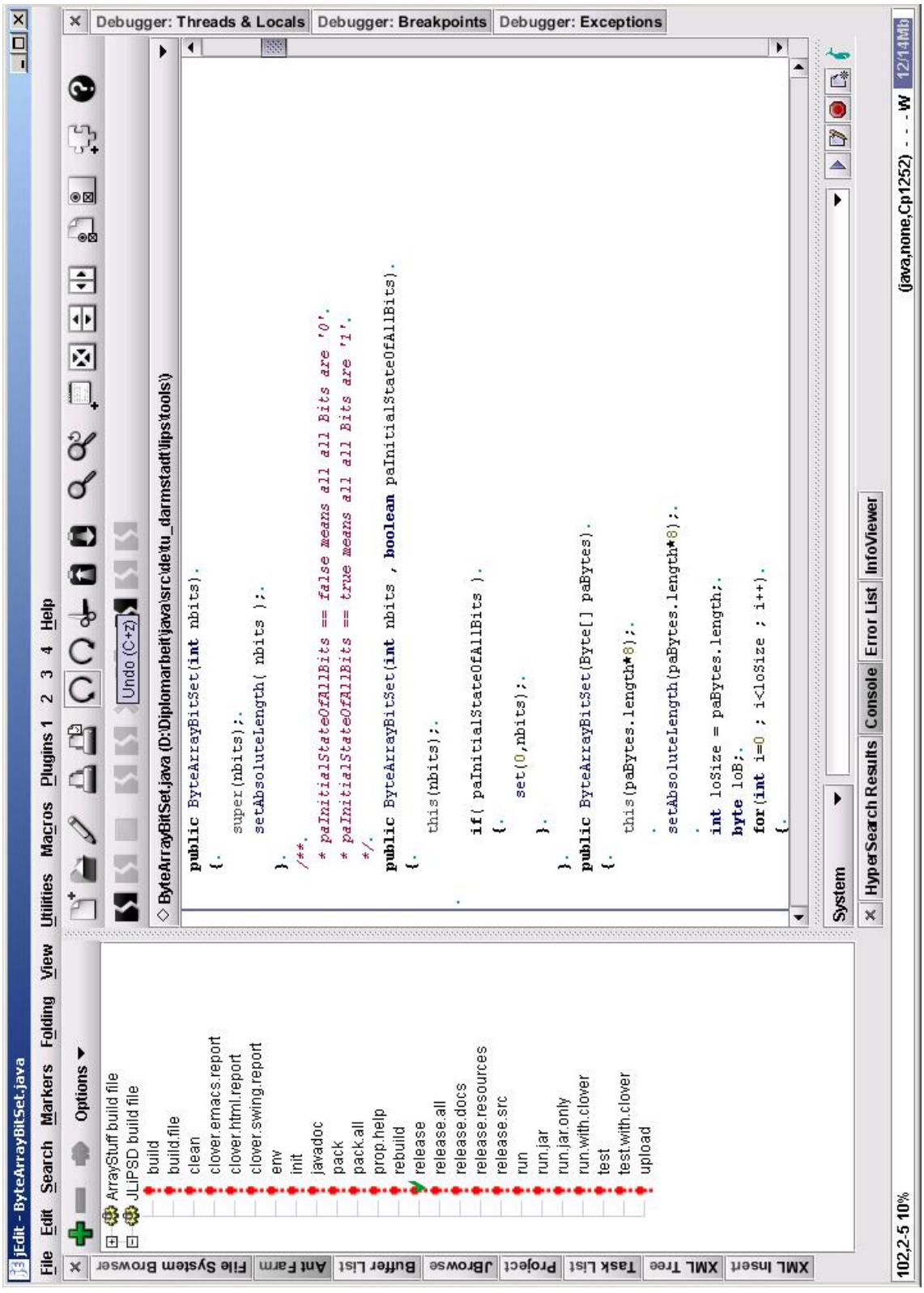

Abbildung 8.3: JEdit mit integriertem Ant-Werkzeug

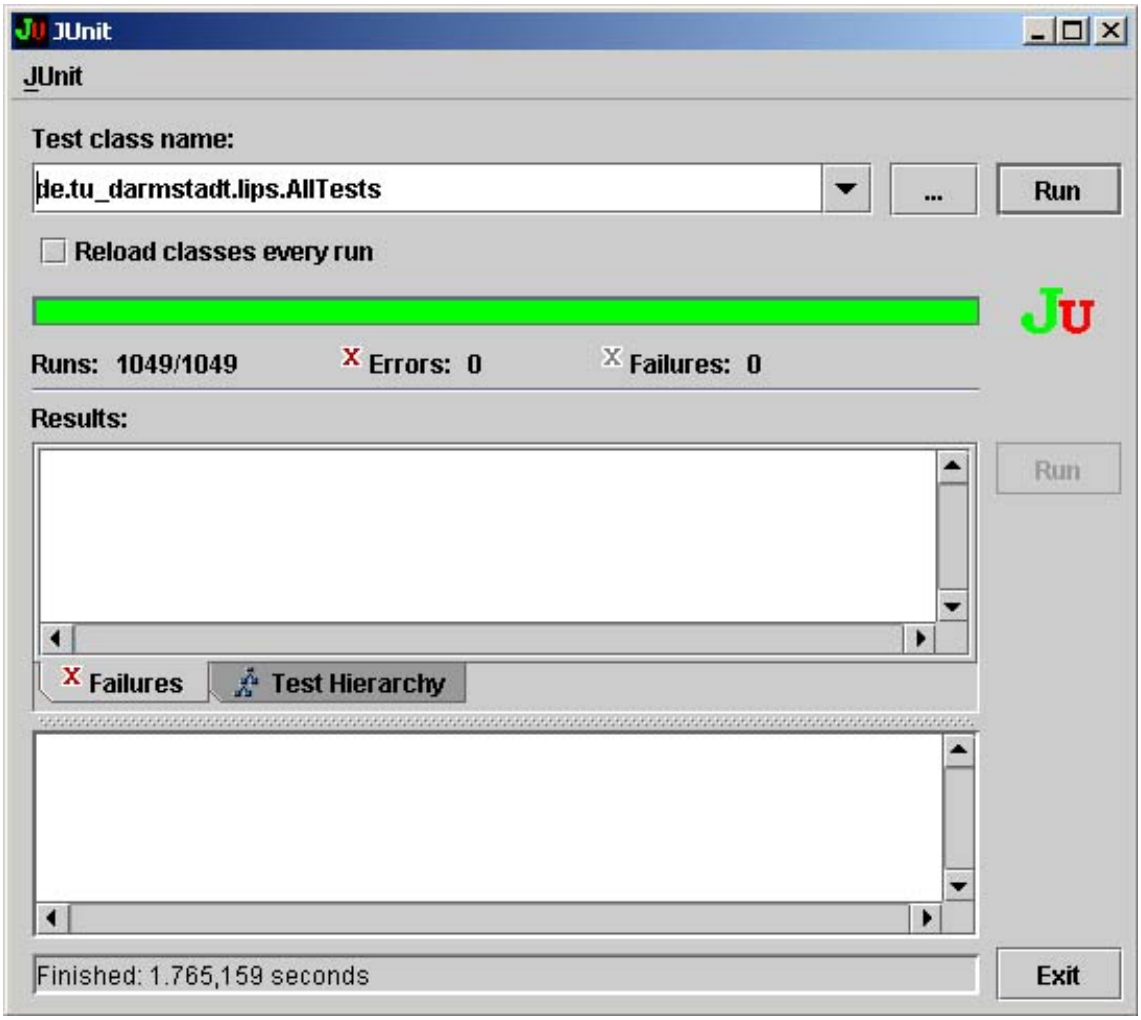

Abbildung 8.4: JUnit : Swing Ergebnisreport

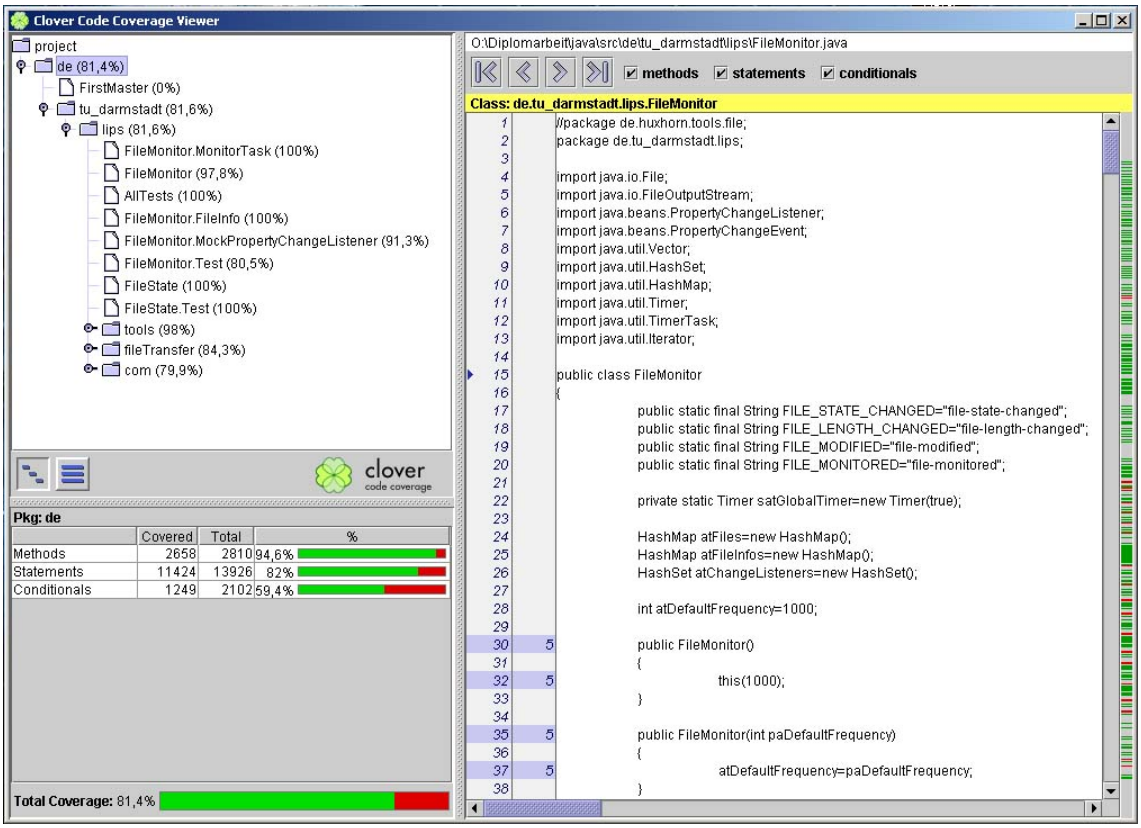

Abbildung 8.5: Clover Swing Report

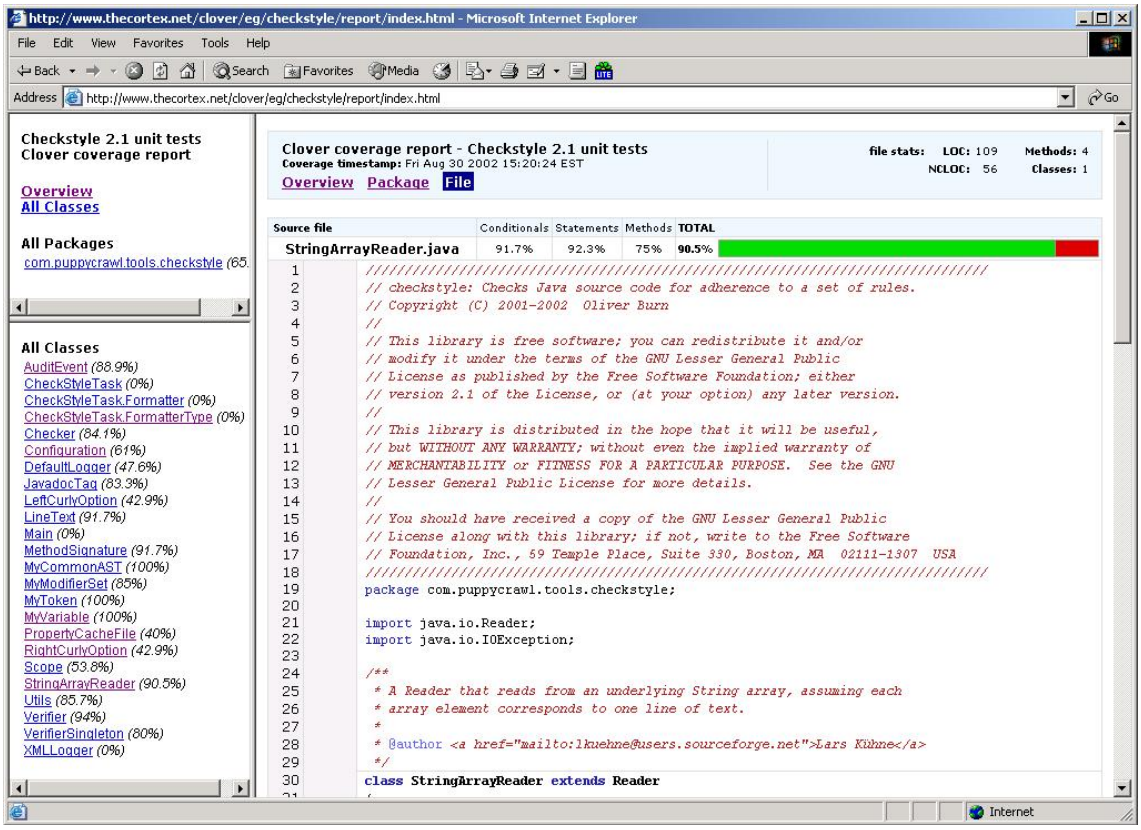
http://www.thecortex.net/clover/eg/checkstyle/report/index.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://www.thecortex.net/clover/eg/checkstyle/report/index.html
Checkstyle 2.1 unit tests
Clover coverage report
Overview
All Classes
All Packages
com.puppycrawl.tools.checkstyle (65.
All Classes
AuditEvent (88.9%)
CheckStyleTask (0%)
CheckStyleTask.Formatter (0%)
CheckStyleTask.FormatterType (0%)
Checker (84.1%)
Configuration (61%)
DefaultLogger (47.6%)
JavadocTag (83.3%)
LeftCurlyOption (42.9%)
LineText (91.7%)
Main (0%)
MethodSignature (91.7%)
MyCommonAST (100%)
MyModifierSet (85%)
MyToken (100%)
MyVariable (100%)
PropertyCacheFile (40%)
RightCurlyOption (42.9%)
Scope (53.8%)
StringArrayReader (90.5%)
Utils (85.7%)
Verifier (94%)
VerifierSingleton (80%)
XMLLogger (0%)
Clover coverage report - Checkstyle 2.1 unit tests
Coverage timestamp: Fri Aug 30 2002 15:20:24 EST
file stats: LOC: 109 Methods: 4
NCLOC: 56 Classes: 1
Overview Package File
Source file Conditionals Statements Methods TOTAL
StringArrayReader.java 91.7% 92.3% 75% 90.5%
1 //
2 // checkstyle: Checks Java source code for adherence to a set of rules.
3 // Copyright (C) 2001-2002 Oliver Burn
4 //
5 // This library is free software; you can redistribute it and/or
6 // modify it under the terms of the GNU Lesser General Public
7 // License as published by the Free Software Foundation; either
8 // version 2.1 of the License, or (at your option) any later version.
9 //
10 // This library is distributed in the hope that it will be useful,
11 // but WITHOUT ANY WARRANTY; without even the implied warranty of
12 // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
13 // Lesser General Public License for more details.
14 //
15 // You should have received a copy of the GNU Lesser General Public
16 // License along with this library; if not, write to the Free Software
17 // Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
18 //
19 package com.puppycrawl.tools.checkstyle;
20
21 import java.io.Reader;
22 import java.io.IOException;
23
24 /**
25 * A Reader that reads from an underlying String array, assuming each
26 * array element corresponds to one line of text.
27 *
28 * @author Lars Kühne
29 */
30 class StringArrayReader extends Reader
Internet

Abbildung 8.6: Clover HTML/XML Report - Klassenansicht

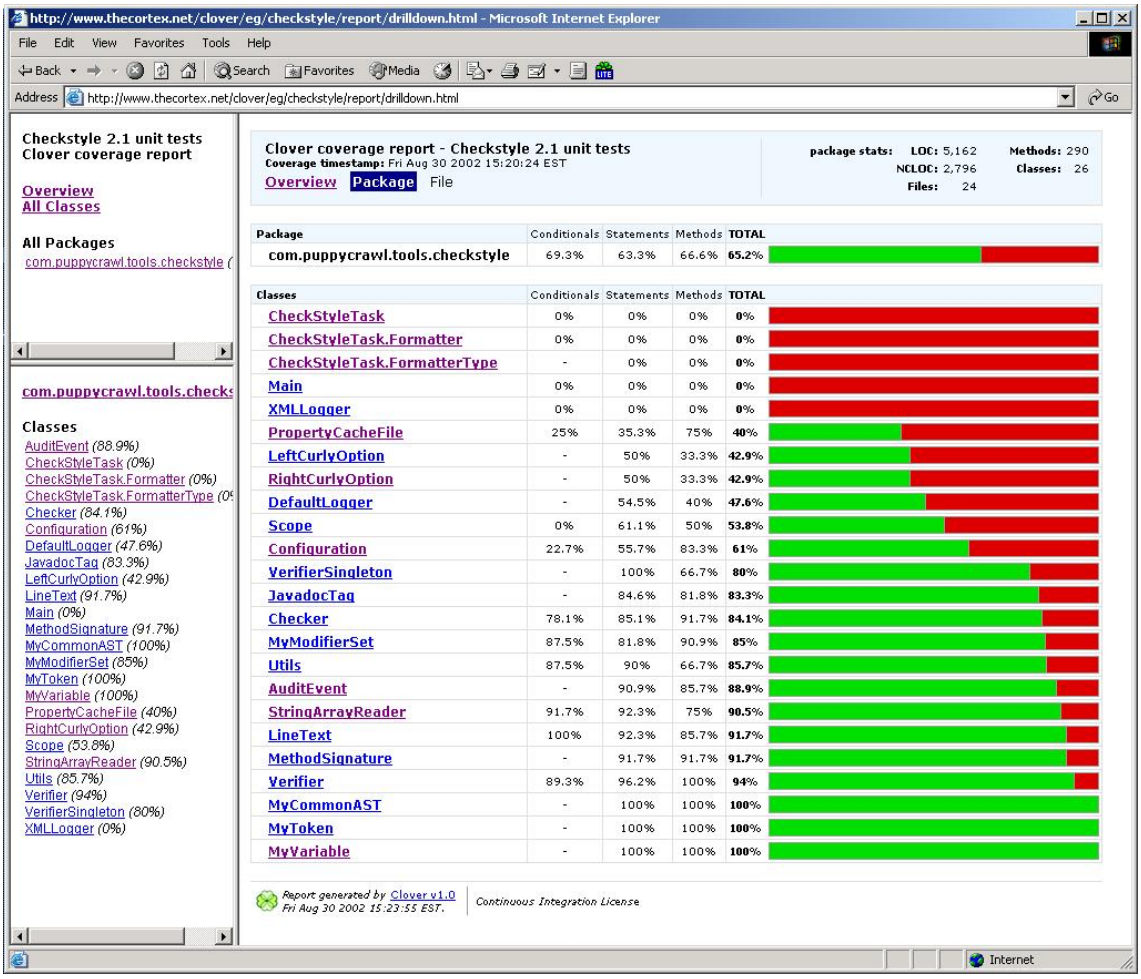

Abbildung 8.7: Clover HTML/XML Report - Testsuiteansicht

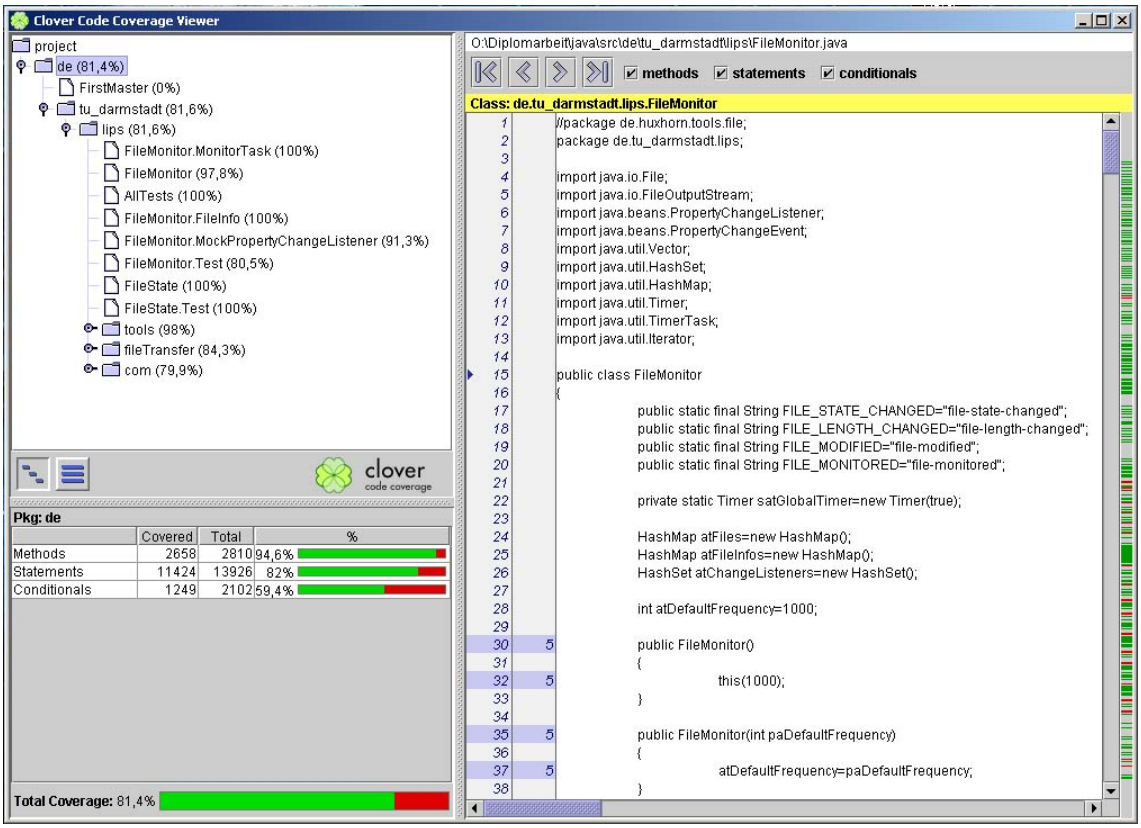

Abbildung 8.8: Clover Überdeckungsergebnis des JLiPSD Projektes

Kapitel 9

Zusammenfassung und Ausblick

Die vorliegende Arbeit hat gezeigt, daß das Testframework für JLiPSD einen hohen Stellenwert bei der Entwicklung des JLiPSD hatte. Dies wird alleine schon anhand der Anzahl der gefundenen Fehler und deren Ursachen deutlich.

Im einleitenden Teil dieser Arbeit wurde gezeigt, welche Stellung das Softwaretesten in der Softwareentwicklung einnimmt. Softwaretesten ist zwingend notwendig um Softwarequalität zu sichern. Dies wurde durch Mechanismen und Verfahren des Softwaretestens erreicht. Hierbei mußten einige Besonderheiten der objektorientierten Programmiersprache Java berücksichtigt werden.

In den weiteren Kapiteln der Arbeit wurde gezeigt, wie die gewonnenen Erkenntnisse sinnvoll auf das Testframework von JLiPSD angewendet werden können. Diese wurden durch ein sinnvolles Design ergänzt. Hieraus wurde, unter Verwendung ausgewählter Werkzeuge, das Testframework entwickelt. Abschließend wurde das Testframework für JLiPSD in seiner Anwendung, sowie dessen Testergebnisse vorgestellt.

Ein wichtiger Aspekt bei der Erweiterung des Testframeworks für zukünftige Projekte wird sein, ob die verwendeten Werkzeuge weiterhin kostenfrei als sogenannte Freeware erhältlich sein werden. Kurz vor Beendigung dieser Arbeit wurde eine neue Version des im Testframework verwendeten Überdeckungswerzeugs Clover veröffentlicht. Die im Testframework für JLiPSD verwendete Clover Version 0.6b steht zwar weiterhin kostenfrei auf der Clover Homepage zur Verfügung, neue Versionen sind jedoch nicht mehr kostenfrei. Als Alternative zu Clover wurden in dieser Arbeit auch andere Überdeckungswerkzeuge reflektiert, welche sich jedoch noch komplett in der Beta Phase ihrer Entwicklung befinden. Desweiteren darf man gespannt sein, wie sich das Softwaretesten allgemein weiterentwickeln wird. Hierbei sei vor allem die Forschung im Bereich der *automatischen Testfallgenerierung* erwähnt.

Ein weiteres Fragezeichen steht auch hinter der Verwendung, beziehungsweise der Zukunft des JLiPSD. Sicherlich wird der JLiPSD an der University of Cairo seine

Verwendung finden, da dort, in Kooperation mit der TU Darmstadt, Teile des LiPS Systems weiterentwickelt werden. Der JLiPSD beziehungsweise das LiPS System wird sicherlich nur dann eine Zukunft haben, wenn es komplett plattformunabhängig betrieben werden kann. Diesbezüglich bleibt nur zu hoffen, daß die Portierung der LiPS Serverseite nach Java irgendwann ihre Umsetzung finden wird.

Literaturverzeichnis

[1] J.Loviscach in: *Absturzgefahr*; c't S. 156 Ausgabe 19/1998.

[2] H.M. Sneed: *Software Qualitätssicherung*; R. Müller,1988.

[3] H. Trauboth: *Software Qualitätssicherung: konstruktive und analytische Maßnahmen*; Oldenburg,1996.

[4] W.S. Humphrey: *Managing the software process*; Addison-Wesley,1989.

[5] H. Balzert: *Lehrbuch der Software Technik - Band 2*; Spektrum, Akademischer Verlag,1998.

[6] W. Perry: *Effective Methods for Software Testing*; Wiley,1995.

[7] Myers, Glenford: *Methodisches Testen von Programmen*; 5. Aufl. München Oldenburg,1995.

[8] Myers, Glenford: *The Art of Software Testing*; Wiley, 1979.

[9] VDI-Gemeinschaftsausschuss Industrielle Systemtechnik: *Softwarezuverlässigkeit*; VDI-Verlag,1993.

[10] C. Kaner, J. Falk, H.Q. Nguyen: *Testing Computer Software*; Int. Thompson Computer Press,1993.

[11] P. Liggesmeyer: *Modultest und Modulverifikation*; BI Wissenschaftsverlag,1990.

[12] M. Deck: *Cleanroom software engineering: Quality improvement and cost reduction*; Pacific Northwest Software Quality Conference,1994.

[13] R.C. Wilson: *Unix Test Tools and Benchmarks*; Prentice Hall,1995.

[14] G. Booch: *Object-Oriented Analysis and Design*; Addison-Wesley,1994.

[15] J. Rumbau, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: *Object-oriented Modeling and Design*; Sams Publishing USA,1991.

[16] B. Bezier: *Testing Technology - The Growing Gap, volume 7*; Cutter Information Corp,April 1994.

[17] K. Beck: *Extreme Programming Explained: Embrace Change*; Addison-Wesley,1999.

[18] J. Link: *Unit Tests mit Java - Der Test-First-Ansatz*; dpunkt Verlag,2002.

[19] S. Edlich: *Ant - kurz&gut*; O'Reilly,2002.

[20] R. Eckstein: *XML - kurz&gut*; 2. Aufl. O'Reilly,2001.

[21] Sitraka: *JProbe Coverage*; http://www.jprobe.com.

[22] K. Beck: *Simple Smalltalk Testing*; Smalltalk Report, Oktober 1994.

[23] K. Beck: *JUnit - Unit testing*; http://www.junit.org.

[24] D. Dixon-Peugh, T. Copeland: *Quilt - JUnit Coverage analysis tool*; http://quilt.sourceforge.net.

[25] P. Browne: *NoUnit - JUnit Test analysis tool*; http://nounit.sourceforge.net.

[26] The Cortex: *Clover - A Code Coverage Tool for Java*; http://thecortex.net/clover.

[27] C. Howells: *Gretel - A Residual Test Coverage Tool*; http://gretel.sourceforge.net.

[28] I. Moore: *Jester - JUnit Test Tester*; http://jester.sourceforge.net.

[29] B. Marick: *The Craft of Software Testing*; Prentice Hall,1995.

[30] B. Beizer: *Software Testing Techniques*; Int. Thompson Computer Press,1990.

[31] R.V. Binder: *Testing object-oriented systems: models, patterns and tools*; Addison-Wesley object technology series,2000.

[32] A. Schneider: *JUnit best practices- techniques for building resilient, relocatable, multihtreaded JUnit*; www.javaworld.com,2000.

[33] A. Müller: *JLiPSD - Eine Portierung des LiPSD nach Java*; TU Darmstadt,2003.

[34] O. Burn: *Chainsaw Tool - The Chainsaw Home Page*; http://logui.sourceforge.net,2002.

[35] O. Burn: *JEdit - Open Source programmer's text editor*; http://www.jedit.orgt,2002.

[36] S. Pestov, D.G. Bobrow, J.L. White: *Clos in context - the shape of the design space. Object Oriented Programming - The CLOS perspective*; MIT Press, 1993.

[37] J. Lippmann: *Integration einer Testumgebung in LiPS*; Universität Saarbrücken, 1997.

[38] Apache Software Foundation: *Log4J Projekt - Apache Jakarta Project*; http://jakarta.apache.org/log4j/docs/, 1999-2002.

Index

Abhaengigkeiten, 22
Ablauffolgen, 37
Ablaufpfade, 37
Abnahmetest, 23, 24, 43
Abstrakte Klassen, 36
Abweichungen, 14, 18
Adressfehler, 82
Aenderbarkeit, 15
Aequivalenzklassenanalyse, 29
Akzeptanztest, 43
Analyse, 15, 17, 80
Anforderungen, 18, 24, 39
Anforderungsdefinition, 24
Anforderungsdokument, 18
Anforderungsphase, 15
Ansatz, 16
Ant, 46, 84
Anweisungen, 11
Anweisungsueberdeckung, 31, 42
Anweisungszeilen, 64
Anwendung, 74
Arbeitsplatzrechner, 8
Architekturen, 46
Architekturunabhaengigkeit, 71
Art und Weise, 19
Assert, 63
Attribute, 36
Aufrufreihenfolge, 57
Aufrufsequenz, 58
Auftraggeber, 16
Ausbaufaehigkeit, 12
Ausfuehrungsprofil, 19
Ausfuehrungsreihenfolge, 38
Ausgabekonsole, 79
Ausgabemoeglichkeit, 78
Ausgabespezifikation, 29
Ausgabewert, 25
Auskommentieren, 76
Ausnahmesituationen, 69, 82
Awt, 78

Basisklasse, 36
Beanshell Scripting, 47
Bedienung, 74
Bedienungsablauf, 65, 68
Bedingungskombination, 31
Bedingungsueberdeckung, 31, 42
Begriffsbildung, 16
Benutzerakzeptanz, 12
Benutzerhandbuch, 53
Benutzerschnittstelle, 33
Benutzerverhalten, 19
Benutzungsanleitung, 11
Benutzungsdokumentation, 10
Benutzungshierarchie, 26
Beobachtungen, 16
Beziehungsgeflecht, 42
Black-Box, 21, 42
Bottom-Up, 22
Budget, 48
Bytecode, 49

Chainsaw Logging Tool, 46
Checkliste, 42
Classloader, 81
Clover, 50, 64, 77, 90
Codierung, 22
Coverage, 45

Dateibearbeitung, 81
Datenstrukturen, 48
Datentypen, 82
debuggen, 38

Definition, 12, 18
Design, 17
Designmuster, 57
Designpatterns, 57
destruktiver Prozess, 19
Diplomarbeit, 48
Dokumente, 23
Downloadmoeglichkeit, 50
Durchschnittswert, 32
dynamisches Binden, 37

Ebenen, 21
Eckdaten, 32
Effekte, 19
Effizienz, 21
Einfachheit, 39
Eingabe, 14
Eingabebereich, 30
Eingabespezifikation, 29
Einheiten, 26
Einsatzbedingungen, 24
Einschraenkungen, 20, 57
Einzelgespraeche, 53
Endprodukt, 16
Entscheidungsinformation, 11
Entscheidungstabelle, 30
Entscheidungsueberdeckung, 31
Entwicklungsdokumentation, 10, 11
Entwicklungsphase, 40
Entwicklungsphasen, 15
Entwicklungsstrategie, 14
Entwicklungsumgebung, 45, 77
Entwurfsphase, 22
Erfahrungswerte, 29
erfolgloser Test, 19
erfolgreicher Test, 18
Erfordernisse, 24
Ergebnisdaten, 78
Ergebnisreport, 49, 51, 65, 78
Erreichbarkeit, 25
Erreichbarkeitsregel, 25
Erstellungsprozess, 16
Erzeugnis, 16
Exception, 42, 69
Exceptions, 82
Exklusion, 51
Extreme Programming, 35, 39

Faeden, 83
Faktoren, 15
Feedback, 39, 41
Fehler, 19
Fehleraufdeckung, 26
Fehlerbedingungen, 24
Fehlerfreiheit, 8, 20
Fehlerkategorien, 31
Fehlerkorrektur, 33
Fehlerquellen, 26
fehlertolerant, 46
Fehlerursachen, 36, 81
Fehlervermeidung, 15
Fehlerwahrscheinlichkeit, 24
Fehlhandlungen, 34
Feind, 14
Feinentwurf, 23
Formulierungsfreiheit, 57
Framework, 36
Funktionalitaet, 16
Funktionalitaetsanforderung, 54
Funktionsabdeckung, 29

Geld, 10
Genauigkeit, 12
Gesamtergebnisse, 79
Gesamtsystem, 24
Grenzen, 20
Grenzwertanalyse, 30
Gretel, 51
Grobentwurfsphase, 22
Grundwerte, 39

Hardwareentwicklung, 22
Hardwarekomponenten, 23
Hauptsuite, 66
Hierarchie, 27
Hierarchieebene, 59
Hilfsprogramme, 38

IBM, 49

ideale Fehlerbedingungen, 24
IEEE, 10
Illustration, 79
Implementierung, 17
Inadaequatheit, 14
Innere Testklassen, 69
Inspektion, 15
Integration, 60
Integrationstest, 26, 43, 44
Interfacetest, 26
Interferenzen, 39
Internetverbindung, 56
Internetverbindungen, 82

Jakarta, 47
Java, 46
JavaWorld, 69
JCover, 48
JDK, 46
JEdit, 46, 74, 84
Jester, 51
JLiPSD, 46, 74, 91
journal file, 33
JProbe, 48
JTest, 48
JUnit, 49, 62
JVerify, 48

Kapselung, 35
Kapselungsprinzip, 36
Kategorie, 16
Kenntnis, 21
Kinderkrankheiten, 50
Klassendiagramm, 62
Klassenhierarchie, 59
Klassenlader, 81
Klassentest, 44, 76
Klassentests, 66
Klassentestwerkzeug, 49, 54, 61
Kodieren, 41
Kodierung, 62
Kommandoprozeduren, 11
Kommandozeile, 64
Kommentare, 53
Kommentarzeilen, 75
kommerziell, 48
Kommunikation, 39
Komplexitaetsmessung, 55
Komponenten, 19, 26
Konstruktion, 25
Kontrollausgaben, 83
Kontrolle, 15
Kontrolleure, 11
Kosten, 16
Kostenentwicklung, 16
kostenfrei, 51
Kriterien, 21
Kundenbeduerfnis, 16
Kundeninteressen, 16

Lasttests, 32
Laufzeit, 37
Lebenszyklus, 14, 17
Lebenszyklustesten, 22
Leistungsdaten, 32
Leistungstests, 32
Lines of Code, 16, 55
LiPS, 46, 91

Maschinenbedarfsanweisungen, 11
Massnahmen, 14
Meetings, 34
Messwerkzeug, 45
Meta-Programme, 38
Methoden, 13, 36, 81
Methodennamen, 68
Metriken, 15, 55
Mindestanforderung, 45
Mock Objekte, 26
Modale Klassen, 58
Modul, 22
Modultest, 26
Motivation, 10
Mut, 39
Mutation Testing Tool, 51

Nachbearbeitungskosten, 16
Namensgebung, 71

Navigation, 78
Netzkapazitaet, 32
Neuentwicklung, 33
Nonmodale Klassen, 57
Notwendigkeit, 13, 25
Notwendigkeitsregel, 25
NoUnit, 49
Nutzen, 16
Nutzergruppe, 19

Oberflaeche, 55
Objektattribute, 59
Objektorientierung, 35, 57
Objektzustand, 57
Open-Source, 49
Orientierung, 48

Packages, 26
Pair Programming, 39
Patternbeschreibung, 58
Patterns, 60
Performance, 47
Pfadabdeckung, 31
Pfadueberdeckung, 42
Phasen, 15
Plaene, 14
Planung, 15, 24
Planungsspiel, 40
Polymorphie, 37
Praedikate, 31
Preis-/Leistungsverhaeltnis, 15
Problemanalyse, 22
Problematik, 57
Problemstellung, 18
Produktionscode, 40
Produktqualitaet, 15
Programmablauf, 19
Programmabschnitte, 83
Programmausschnitt, 37
Programmcode, 13
Programmiereditor, 46
Programmiersprache C, 8
Programmierteam, 19
Programmlogik, 21, 83
Programmteile, 21
Programmverhalten, 21, 42
Projekteinschaetzung, 48
Propagierung, 25
Protokolle, 82
Pruefverfahren, 37

Qualitaetslenkung, 12
Qualitaetsmanagement, 17
Qualitaetsmerkmal, 15
Qualitaetsmerkmale, 11
Qualitaetsplaene, 14
Qualitaetsplanung, 12
Qualitaetsprodukte, 14
Qualitaetspruefung, 12
Qualitaetssicherung, 8
Quasimodale Klassen, 58
Quellcode, 48
Quilt, 50

Rangordnung, 16
Redundanzen, 83
Refactoring, 39
Referenzergebnis, 20
Reflection, 38
Reflectiontesten, 38
Reflektion, 80
Regelwerk, 41
Regressionstests, 33
Reihenfolge, 58
Reinraumprozess, 32
Reinraumtests, 33
Reinstrumentierung, 51
Reportart, 78
Reportmoeglichkeiten, 50, 78
Reproduzierbarkeit, 11
Resourcen, 17
Review Meetings, 34, 42
Richtlinien, 14
Rueckgabewerte, 57

Sandwichtesten, 27
Schaden, 15
Schluesseltechnologie, 47

Schnittstellen, 26, 36
Schriftfuehrer, 34
Schwachpunkte, 18
Schwachstellen, 24
Seiteneffekte, 70
Serverseite, 46
Sicherheit, 24
Simulation, 24
Smalltalk, 49
Software Lebenszyklus, 14, 22
Softwaredesign, 40
Softwareentwicklung, 10
Softwareentwicklungsprozess, 13
Softwareentwurf, 22
Softwarefehler, 13
Softwarekomponente, 18
Softwaremanager, 13
Softwaremetriken, 15
Softwareprodukt, 14
Softwarequalitaet, 11
Softwarequalitaetssicherung, 10
Softwaresystem, 11, 18, 26
Softwaretechnik, 10
Softwaretesten, 17, 18
Speicher, 32
Spezifikation, 18, 22, 30
Standards, 13
Steuerung, 15
Steuerungssoftware, 22
Strategie, 24
Stress, 10
Stresstests, 32
Strukturelemente, 30
Strukturtest, 21
Stubs, 26
Stylephase, 54
Subsysteme, 23
Suite, 63
Supportverzeichnisse, 54
Swing, 64
Systementwurf, 24
Systemkomponente, 21
Systemskalierung, 24, 25
Systemtest, 26, 43
Systemtests, 71
Systemzustaende, 56

Taetigkeiten, 12
Taetigkeitsbereich, 13
Techniken, 69
Teilaufgaben, 27
Test-First Ansatz, 41
Testabdeckung, 21
Testaktivitaeten, 28
Testanforderung, 53
Testanpassungen, 54
Testart, 20
Testaufwand, 34
Testausfuehrung, 77
Testauswertung, 54, 55
Testbeschreibung, 75
Testbezeichner, 56
Testcase, 63
Testdaten, 21
Testdatenkombination, 27
Testdokumentation, 11, 45
Testdokumente, 54
Testen, 15, 18, 41
Testentwickler, 26
Testergebnisse, 55, 78
Testfaelle, 67
Testfallbeschreibung, 56
Testfallermittlung, 24, 29
Testfallnamensgebung, 57
Testfallverifikation, 56
Testframework, 45, 53
Testguete, 21
Testhierarchie, 36
Testlauf, 55, 79
Testmesswerkzeug, 45
Testmethode, 56
Testmethoden, 32
Testmethodik, 41
Testmuster, 57, 60
Testobjekt, 25, 27
Testpatterns, 57
Testphase, 15, 22, 54, 55

Testplanung, 54
Testprozess, 17, 26, 53
Testreport, 78
Testressourcen, 24
Testspezifikation, 53, 56
Teststufen, 43
Testsuite, 44, 76
Testsuiten, 54
Testtreiber, 36
Testueberdeckung, 21
Testumgebung, 62
Testverfahren, 16, 28, 31
Testwerkzeug, 45, 51
Textausgabe, 78
threads, 83
TomCat, 49
Top-Down, 21

Ueberdeckung, 21
Ueberdeckungsergebnisse, 80
Ueberdeckungskriterien, 42
Ueberdeckungsluecke, 68
Ueberdeckungsmonitore, 51
Ueberdeckungsrate, 53, 69
Ueberdeckungstestwerkzeug, 51
Ueberdeckungswerkzeug, 49, 64
Ueberschaubarkeit, 12
Umgebung, 24
Unimodale Klassen, 58
Unittest, 26, 43
University of Cairo, 90
Unix, 46
Unterklassen, 59
Unterpunkt, 16
Untersuiten, 54
Unvollstaendigkeiten, 18
Ursache-Wirkungs-Analyse, 30
Ursachen, 30
User Story Cards, 40

V-Modell, 27
Validation, 17
Variablen, 56
Variablenzugriff, 31
Variationsmoeglichkeiten, 20
Veraenderbarkeit, 12
Verbindungsfehler, 82
Vererbung, 36
Vergleiche, 82
Vergleichssituationen, 82
Verifikation, 15, 17, 56
Verifikation ohne Validation, 53
Verlaesslichkeitstesten, 32
Versionsplan, 40
Verstaendlichkeit, 11
Vertrauen, 18
Vertreterfunktionalitaet, 21, 56
Verwendungsfehler, 83
Vollstaendigkeit, 12
Vorgehensmodell, 14
Vorgehensweise, 22

W3C, 47
Wahrnehmungen, 16
Wahrscheinlichkeit, 24, 30
Warteschlange, 58
Wartungspersonal, 11
Weiterentwicklung, 48
Werkzeuge, 23
Werkzeugunterstuetzung, 31
White-Box, 21, 42
Widerspruchsfreiheit, 12
Wiederverwendung, 36

XML, 46
XP, 35, 39
XSL-Stylesheets, 48

Zeichenkette, 20
Zeit, 10
Zeitersparnis, 39
Zeitrahmen, 40
Zeitunabhaengigkeit, 71
Zielvorgaben, 19
Zugriffsrechte, 67
Zustandsuebergangsgraph, 43
Zustandsuedergangsgraph, 59
Zweigueberdeckung, 31, 42
Zwischenprodukt, 14

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, März 2003 Jochen Hähnle